Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky

Diplomová práce

Přibližné vyhledávání nad vlastními indexy

Bc. Lukáš Hrbek

Vedoucí práce: prof. Ing. Jan Holub, Ph.D.

4. května 2015

Poděkování

Rád bych poděkoval svému vedoucímu práce za nabídku tématu s poměrně širokým využitím v praxi. Také bych chtěl poděkovat všem, kteří mě podporovali během jeho vypracování.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, avšak pouze k nevýdělečným účelům. Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 4. května 2015 .....

Odkaz na tuto práci

Hrbek, Lukáš. *Přibližné vyhledávání nad vlastními indexy.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Práce se zaměřuje na přibližné vyhledávání s nejvýše k chybami. Chyby jsou definovány s využitím Levenshteinovy vzdálenosti. Pro řešení této úlohy jsem navrhl filtrační algoritmus založený na pigeonhole principu. Prohledávaný text se předpokládá velký, implementovaný algoritmus proto používá FM-Index. Program je na poli vyhledávání v DNA srovnán s nástrojem BLAST. Experimenty ukázaly, že v některých aspektech je má implementace lepší.

Klíčová slova Filtr, přibližné vyhledávání, DNA, index

Abstract

The work focuses on the approximate string matching with no more than k differences. Differences are defined by Levenshtein distance. I designed for the solution of this task filtering algorithm based on the pigeonhole principle. The text is assumed to be large, implemented algorithm therefore uses FM-index. The program was in the field of searching in DNA compared with the BLAST tool. Experiments have shown that in some aspects is my implementation better.

Keywords Filter, approximate string matching, DNA, index

Obsah

Seznam obrázků

Úvod

Vyhledávání je jednou základních aplikací výpočetní techniky. Uživatelé běžně zadávají vyhledávací *dotazy*, ať profesně speciální povahy či jako orientaci v prostoru plném informací. Vyhledávání v datech je také prováděno mnohými algoritmy jako vedlejší operace. Pokud budeme na data pohlížet jako na text složený ze symbolů z pevně určené množiny, tedy *abecedy*, můžeme mluvit pouze o vyhledávání v *textu*, přestože takový text může být reprezentací zcela jiných dat (stejně tak symboly je možné interpretovat různě – např. jako znaky, nebo jako bajty, či sekvence bajtů).

Základní funkcí vyhledávání je nalezení místa výskytu zadané fráze, tedy *vzorku* textu. V rozhodovací podobě stačí jako odpověď oznámení, zda je vzorek v textu přítomen. Na druhé straně škály je enumerační formulace zadání, která vyžaduje jako odpověď úplný seznam všech *výskytů* zadaného vzorku ve zkoumaném textu. Tato formulace je nejobecnější, proto se budu věnovat jí. Řešení tohoto problému odpovídá také na otázky méně obecné (jako například nalezení prvního výskytu, alespoň jednoho výskytu, nebo jen počtu výskytů).

Bylo vyvinuto mnoho metod jak provádět přesné vyhledávání. Ovšem skutečné potřeby, zvláště dotazy běžných uživatelů, často vycházejí pouze z matného určení toho, co je třeba nalézt. Hledaný objekt se může ve zdroji objevovat v jiné podobě, než jak jej zadávající specifikoval a přesto je relevantní. Ať je odlišnost způsobená chybou zadávajícího (tedy ve vzorku), či chybou v prohledávaném textu a nebo jsou drobné rozdíly zcela přirozené (jako například při skloňování), přesné vyhledávání není možné pro tyto účely využít. Za výjimku lze považovat případ, kdy je možné jednoznačně zúžit abecedu bez ztráty významu symbolu – tak je možné vyhledávat v přirozeném jazyce také výskyty lišící se jen velikostí písmen. Obecně je však nutné považovat každý rozdíl za *chybu*, protože například v matematickém textu pravděpodobně 'F' označuje jiný objekt než 'f'.

Příklady výše ilustrují, že vyhledávání s chybami je poměrně přirozené, ovšem může se různit definice toho, co je chyba. Bylo proto nutné zabývat se novým problémem nazvaným *přibližné* vyhledávání, který dle určité definice

chyby rozšíří výsledky vyhledávání na všechny relevantní výskyty i bez nutnosti dekódovat sémantiku textu. Nastavením pak může být upřesněno, o jak velké rozšíření může jít (typicky maximální počet tolerovaných chyb). Tato práce se bude dále věnovat vyhledávání výskytů, jejichž podobnost se vzorkem se dá stanovit standardními metrikami vzdálenosti dvou *řetězců* (v tomto případě vzorku a části, neboli podřetězci, textu).

Druhým aspektem je v zásadě stále rostoucí množství dat, která jsou zpracovávána. Toto zpracování se samozřejmě nevyhýbá ani vyhledávání. Jistým rozdílem je však to, že vyhledávání často určuje, které části původního objemu budou zpracovány, či jestli vůbec mají být data zpracovávána. Aby nemusel být zdlouhavě prohledáván celý objem dat (zvláště pokud vyhledávání ukáže, že není třeba dalšího zpracování), je možné obsah textu *indexovat* a urychlit tak přístup k určitým jeho částem. Extrémně dlouhé texty je také vhodné uchovávat v *komprimované* podobě a šetřit tak místo úložiště. Vlastní index (neboli *self-index*) nabízí obě služby najednou, navíc lze očekávat možnost dekomprimovat za účelem zpracování jen část původních dat. Tato práce se nezabývá návrhem vlastního indexu, ale předpokládá využití služeb, které vlastní indexy nabízejí ke složitějšímu způsobu vyhledávání. Protože schopnost komprese není nutnou podmínkou, budu dále používat jen označení „index“. Indexem bude, vzhledem k charakteru práce, myšlen indexovací algoritmus, nikoli indexový soubor.

Kapitola 1

Motivace

1.1 Prohledávání DNA

Přestože je má práce obecnější, představím nyní problém, jímž byla motivována. Pro snazší vyjádření se i v pozdějším textu budu vracet k tomuto problému jako k příkladu a terminologie může být přizpůsobena problematice práce s DNA. Také bude představeno značení veličin, které odpovídá konvencím pro vyhledávání v textu. Tento motivační problém zjevně poukazuje na jednu z možností praktického využití, zároveň svými parametry výstižněji vysvětluje, proč se návrh ubírá daným směrem.

Bioinformatika je vědní disciplína zabývající se mj. analýzou molekulárně biologických dat, typicky tedy DNA. Výzkum DNA je významný zejména pro různá odvětví medicíny, ačkoliv jeho dopad není omezen jen na lékařství. Základním úkonem bývá analýza podobnosti jistých částí dvou molekul (DNA, či bílkoviny). Tedy přibližné vyhledání sekvence vzorku *P (Pattern)*, v dlouhé sekvenci textu *T (Text)*, jímž je často DNA. Takto lze vyhledávat a mapovat určité geny, i když často mutují a samy techniky čtení genomu zavádějí další chyby.

Sekvence DNA se skládá z jednotlivých *bází*: adeninu, thyminu, guaninu a cytosinu. Ty jsou typicky reprezentovány jako znaky odpovídající prvnímu písmenu daného nukleotidu. Jde o velmi malou abecedu $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, v ideálním případě jen s velikostí $\sigma = 4$, ovšem díky nejednoznačnostem při čtení molekuly se v praxi používá symbolů mnohem více (organizace *IUPAC* proto definuje pro DNA celkem 16 symbolů).

Značnou překážkou je přitom velikost genomu, která je v případě (haploidního[1]) lidského genomu[2] zhruba *3,2 Gbp* (miliard bází) [1]. Tedy $|T| = 3,2{\cdot}10^9$, což bez použití komprese znamená zpracovávat 3GB soubor. Taková databáze je proto většinou zpracována vlastním indexem.

[1] Haploidní buňky mají 23 chromozomů, diploidní buňky však mají 46 chromozomů.

[2] Mimo jaderné DNA se v lidských buňkách nachází také mitochondriální DNA, kruhová molekula délky lehce přes *16,5 kbp*.

Vyhledávaný vzorek bude sekvence využívající stejné abecedy s délkou[3] řekněme $|P| = 50$. Velikost vzorku se také označuje jako m, obdobně se při vyjádření složitosti běžně používá n místo $|T|$ (a také $\sigma = |\Sigma|$). Při popisu algoritmů je toto značení zachováno i v případě, že jsou používány k řešení *dekomponovaného* problému (menších instancí stejného problému).

Dále je nutné určit, s jakou přesností chceme vzorek vyhledat. Často se stane, že DNA je modifikována vyříznutím jisté části, či jejím vložením. Při manipulaci (typicky při replikaci) se vyskytnou mutace jednotlivých bází, a to zhruba v 1% případů [2]. Tyto druhy chyb odpovídají chybám v editační vzdálenosti. Stanovme počet chyb například na $k = 3$, což znamená, že za výskyt vzorku považujeme sekvenci v textu lišící se od vzorku nejvýše ve třech (editačních) chybách. Pochopitelně že pro větší velikosti vzorku lze předpokládat úměrně větší počet chyb, proto je možné míru přesnosti, resp. nepřesnosti, také vyjádřit *chybovým poměrem* rovnicí 1.1. Lze vysledovat, že pro práci s DNA se používá okolo $\alpha = 10\%$.

$$\alpha = \frac{k}{m} \tag{1.1}$$

1.2 Požadavky na realizaci

Cílem praktické části práce je program pro přibližné vyhledávání s max. k chybami, danými Hammingovou (popř. editační) vzdáleností. Vstupní text a vzorek by měl být zadán souborem, protože algoritmus by neměl být omezen abecedou tisknutelných znaků. Součástí programu bude také index (existující algoritmus), jenž by měl nabízet operaci vytvoření a uložení komprimovaného a indexovaného souboru textu. Pro vyhledávání pak bude vyžadován tento soubor, nikoli však už originální soubor textu. Tento požadavek umožňuje distribuovat znatelně menší soubory, ideálně dostatečně malé aby mohl být celý genom načten do operační paměti osobního počítače. Součástí zadání mé diplomové práce je požadavek na implementaci nad *FM-Indexem* [3], avšak návrh algoritmu by měl být dostatečně obecný, aby bylo možné použít jiný index (splňující požadované rozhraní), není přitom podmínkou, aby nutně umožňoval kompresi či indexování (pro malé soubory by to nemuselo být výhodné).

Na rozdíl od běžně používaných programů na prohledávání DNA (viz sekce 2.4) se v návrhu snažím zachovat bezztrátovost. Nalezeny by tedy měly být všechny výskyty a přitom každý výskyt by měl být hlášen pouze jednou. Požadavky předešlé věty, ač logické, mohou být různě vyloženy pro tzv. indukované výskyty. Příkladem budiž výskyt, který je *vlastním podřetězcem* jiného výskytu. V takovém případě je z praktických důvodů doporučeno ohlásit jen jeden výskyt z této množiny. Dále poznamenejme, že prázdný řetězec by neměl být považován za výskyt, samotné vyhledávání prázdného řetězce totiž postrádá smysl.

[3] Za krátké vzorky můžeme považovat sekvence do 20bp, dlouhé v řádu stovek bp.

Dle volby uživatele by mělo jít o vyhledávání s chybami v Hammingově, nebo Levenshteinově vzdálenosti. Případně přesné vyhledávání jako speciální případ. Výsledkem jsou pak všechny přesné výskyty a všechny výskyty s nejvýše k chybami. Zadané k by mělo být relativně malé[4] přirozené číslo.

Pro každý výskyt by mělo být ve výstupních informacích uvedeno, kde se v originálním textu nachází (např. pozice počátku a konce) a počet chyb, tedy vzdálenost od vzorku. Pro praktické využití je dále vhodné zobrazit výskyt a znázornit, na kterých jeho místech se nacházejí jaké chyby. Tato operace je také známa jako *zarovnání*. Pokud optimální zarovnání není jednoznačné, stačí (v duchu myšlenky z předchozího odstavce) zobrazit pouze libovolné jedno z nich.

Předpokládá se, že program bude mít možnost další parametrizace (např. volbu algoritmu). Při vhodném nastavení se předpokládá relativně malá paměťová náročnost a dostatečná rychlost (samozřejmě úměrná požadavku). Program by měl fungovat na PC s OS *GNU/Linux*, jenž je v bioinformatice běžnou platformou, portabilita na jiné systémy či architektury je vítána.

1.3 Struktura práce

V úvodu práce a v následující sekci 1.1 je popsán řešený problém. Během popisu jsou vysvětleny později používané termíny a značení. Úvod popisuje téma práce obecně, zatímco v kapitole „Motivace“ jsou podrobnosti vysvětleny na prohledávání DNA. Ve zbytku práce se nadále vracím k této motivační úloze, přestože vyhledávání v DNA není jedinou aplikací pro vytvořený program. Co by měl program vykonávat je uvedeno v sekci 1.2.

V kapitole „Rešerše“ jsou představena různá existující řešení. Sekce 2.1 představuje základní rozdělení algoritmů a zmiňuje významné algoritmy daných druhů. V sekci 2.2 jsou přiblíženy důležité metriky a další dále používané pojmy. Během kapitoly jsou zdůvodněny základní rozhodnutí návrhu a text práce se dále soustředí na zvolený přístup. Proto jsou v sekci 2.3 podrobněji rozebrány filtrační algoritmy a představeny různé přístupy, jenž jsou později často zmiňovány.

V sekci 2.4 je podrobně přiblíženo existující řešení pro zkoumaný problém. Program BLAST je specializovaný na prohledávání DNA, jenž se stalo motivací práce. Popsány jsou základní vlastnosti BLASTu a stručně přiblížen způsob jakým je realizován.

V kapitole „Analýza a návrh“ jsou jednotlivé aspekty problému řešeny podrobněji. Vysvětleny jsou další pojmy a navrhovány řešení popisovaných situací. V této kapitole jsou také uvedeny různé vzorce a interpretován jejich význam. Po zmapování složitého prostoru parametrů je v sekci 3.5 navržen zobecněný filtrační algoritmus a představen jeho jednoduchý matematický mo-

[4] Předpokládá se číslo menší než $|P|$, maximálně však v řádu tisíců.

del. Tento model je následně prostřednictvím grafů použit k odhadu chování implementovaného algoritmu.

Kapitola „Realizace“ představuje v sekci 4.1 návrh modulární struktury mého řešení. V následujících sekcích jsou pak přiblíženy jednotlivé moduly. Představeny jsou dílčí problémy a realizované i nerealizované návrhy jejich řešení. Součástí popisu různých vylepšení je také vysvětlení nově zavedených pojmů. Implementované algoritmy jsou popsány velmi stručně, nechybí však vyjádření časové a paměťové složitosti.

Kapitola „Experimenty“ nejdříve v sekci 5.1 informuje, jak bylo měření prováděno. V dalších sekcích jsou uvedeny nejdříve použitá data, jejich zdroje a vlastnosti. Následně jsou popisovány jednotlivé experimenty a interpretovány výsledky presentované na grafech. Sekce 5.2 je zaměřena na podrobné měření implementovaného řešení, včetně ověření přínosu realizovaných vylepšení. Zbývající vlastnosti jsou měřeny v sekci 5.3, kde je program srovnáván s posledními verzemi BLASTu.

V závěru práce je pak shrnuto zhodnocení realizace.

Kapitola 2

Rešerše

2.1 Základní druhy algoritmů

Přibližné vyhledávání (*approximate string matching*) je problém, jehož složitost se odvíjí od toho, jakým způsobem jsou definovány chyby, které vyhledávání dovoluje. Podle toho může být řešení lineární až NP-úplné [4].

Nejčastěji se používá definice dle Levenshteinovy vzdálenosti (*edit distance*[5]), toto pojetí je dostatečné pro většinu aplikací a použité algoritmy většinou není těžké upravit na odlišnou metriku. Přibližné vyhledávání v Levenshteinově (či editační) vzdálenosti se také nazývá „string matching with k differences“.

Dále je mnoho pozornosti věnováno zjednodušení problému, které výskyty posuzuje podle Hammingovy vzdálenosti. Tato verze problému bývá označována jako „string matching with k mismatches“. Proto se budu nadále zaměřovat především na tyto metriky vzdálenosti.

2.1.1 Dynamické programování

Nejstarší oblastí algoritmů pro přibližné vyhledávání je tzv. dynamické programování (*DP*). V této oblasti bylo učiněno nejvíce teoretických průlomů ve zlepšení nejhoršího případu, přesto v praxi málokdy patří k nejrychlejším.

Základem je algoritmus pro výpočet vzdálenosti dvou řetězců pomocí matice DP. Pokud si představíme text vodorovně a vzorek svisle, matici je možné vyplňovat po sloupcích nebo po řádcích. V posledním řádku je pak možné lokalizovat výskyty podle vypočtené minimální vzdálenosti na dané pozici (výskyt je platný pro hodnoty $\leq k$). Významnou výhodou je pak možnost použít různé metriky pro výpočet vzdálenosti.

Pro další informace k základním DP algoritmům viz sekce 2.2. Z popisu, který jsem uvedl, vyplývá paměťová složitost $\mathcal{O}(m \cdot n)$, ale mnozí si nezávisle uvědomili, že pokud je k vyplnění nového sloupce resp. řádku potřeba jen

[5] Váhy operací jsou standardně rovny 1 (*simple edit distance*), alternativou je vážená verze (*general edit distance*).

jeden předchozí sloupec resp. řádek, pak postačí uchovávat jen tyto dva. Protože vzorek je běžně několikanásobně menší než text, je možné více ušetřit postupem po sloupcích. Paměťová složitost je pak $\mathcal{O}(m)$.

Co se týká časové složitosti, největším objevem jsou tzv. *diagonální* algoritmy. E. Ukkonen si v roce 1983 uvědomil, že diagonály jsou monotónně rostoucí, navíc rostou vždy o konstantu [5]. Tohoto faktu bylo později mnohokrát využito v dalších algoritmech a vylepšeních. Jedním z nejzajímavějších (především pro bioinformatiku) je pak Myersův algoritmus z roku 1986 (otištěn byl však až v roce 1998, jako součást větší práce). Algoritmus je schopen nejen nalézt pozice, ale také řetězce výskytů v čase $\mathcal{O}(k \cdot n)$ (s paměťovou náročností $\mathcal{O}(n)$) [4]. Algoritmy podobné Myersovým (z výše zmíněné práce) v roce 1989 také navrhli Galil a Park [4].

O první významné vylepšení průměrného případu se opět zasloužil Ukkonen v roce 1985 algoritmem, jenž je nyní znám jako „cut-off heuristika“. Hlavní myšlenkou vylepšení spočívá v tom, že buňka s hodnotou větší než $k+1$ se již nemůže podílet na výsledku. Průměrná časová složitost je $\mathcal{O}(k \cdot n)$ a paměťová $\mathcal{O}(m)$ [4].

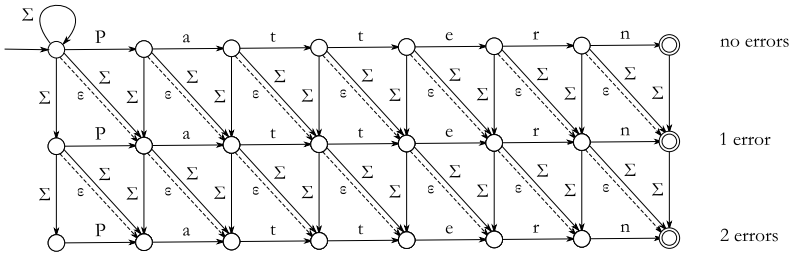
V praxi nejrychlejším algoritmem v oblasti DP je pak vylepšení tohoto algoritmu, které uskutečnili Chang a Lampe v roce 1992 [4]. Vylepšení je založeno na existenci rostoucích podsekvencích čísel ve sloupcích, přičemž prvky sledovaných podsekvencí rostou právě o 1. Průměrná délka těchto sekvencí je $\mathcal{O}(\sqrt{\sigma})$ a průměrná časová složitost tak byla vylepšena na $\mathcal{O}(\frac{k \cdot n}{\sqrt{\sigma}})$. Avšak protože je vylepšení založeno na jednotkových cenách, dá se na rozdíl od ostatních algoritmů těžko přizpůsobit jiným metrikám.

2.1.2 Automaty

Konečné (deterministické) automaty jsou také poměrně starou oblastí. Avšak algoritmy tohoto typu mají nejlepší složitost pro nejhorší případ, a sice pouze $\mathcal{O}(n)$. Na druhou stranu je jejich nevýhodou exponenciální časová i paměťová složitost vzhledem ke k, potřebná pro vytvoření a uchování automatu.

Základem je vyhledávací *nedeterministický konečný automat* (NFA) který přijímá vzorek a jeho okolí specifikované definicí vzdálenosti a maximálním počtem chyb. Takovým automatem je nakonec zpracován celý text, přičemž koncový stav signalizuje na dané pozici výskyt.

Aby bylo možné takovou představu realizovat, je třeba převést NFA na *deterministický konečný automat* (DFA). Konstrukce DFA se však ukazuje jako hlavní problém. Proto Kurtz v roce 1996 navrhl tzv. línou konstrukci automatu. Tato technika za běhu automatu konstruuje jen stavy a přechody, jenž jsou pro zpracování textu nezbytné. Paměťovou složitost tak dokázal redukovat v nejhorším případě na $\mathcal{O}(m \cdot n)$ [4]. Alternativou je pak místo převodu simulace běhu NFA.



Obrázek 2.1: NFA pro vyhledání vzorku „Pattern“ s $k = 2$.

2.1.3 Bitový paralelismus

Tento druh algoritmů je relativně nový a zabývá se jistým druhem paralelizace jiného algoritmu. Tento druh paralelizace není založen na vícevláknovém pojetí úlohy, ale naopak na úrovni bitů. I nejjednodušší procesor totiž bity zpracovává paralelně, totiž po slovech. Označme tedy velikost slova v bitech w.

Značného zrychlení je možné dosáhnout při vyhledávání krátkých vzorků. Počet chyb k navíc nezhoršuje efektivitu [4]. Hlavními trendy jsou paralelizace výpočtu matice dynamického programování, anebo paralelizace simulace běhu nedeterministického konečného automatu.

V případě bitové paralelizace DP se zdá být nejlepší Myersův algoritmus z roku 1998 se složitostí $\mathcal{O}(\lceil \frac{m}{w} \rceil \cdot n)$, v průměrném případě dokonce $\mathcal{O}(\lceil \frac{k}{w} \rceil \cdot n)$ [4]. Tyto algoritmy je však obtížné přetvořit pro jiné metriky vzdálenosti.

Při bitové paralelizaci NFA se v roce 1996 Baeza-Yates a G. Navarro inspirovali dynamickým programováním a o tři roky později nalezli diagonální[6] vylepšení paralelizace NFA. Časovou složitost má v nejhorším případě $\mathcal{O}(\lceil k \cdot \frac{m-k}{w} \rceil \cdot n)$, průměrně pak $\mathcal{O}(\lceil \frac{k^2}{w} \rceil \cdot n)$. Výhodou tohoto algoritmu je, že editační operace mohou mít různé (celočíselné) ceny a podporuje také rozlišování různých tříd symbolů.

2.1.4 Filtrační algoritmus

Tato oblast algoritmů je také mladá a také využívá ke své funkci jiných algoritmů. Principem *filtračních algoritmů* je neprohledávat oblasti textu, jenž neobsahují žádný výskyt. Průměrná složitost je teoreticky $\mathcal{O}(n \cdot \frac{k+\log_\sigma(m)}{m})$, která, jak bylo dokázáno, je optimální [4]. A také v praxi jsou filtrační algoritmy nejrychlejší [4]. Společnou nevýhodou je pak limitace chybovým poměrem (pro vysoké hodnoty α je filtrace velmi neefektivní, ale pro nízké je tomu zase naopak).

Základní myšlenka je založená na tom, že je mnohem snazší výskyty vyloučit než naopak. Filtrační algoritmus proto není schopen sám výskyty najít, k tomu využívá jiný vyhledávací algoritmus. Většina filtrů pak těží z toho, že

[6] Jde o diagonály na grafu NFA, obsahující ε-přechody.

využívá algoritmy pro přesné vyhledání, které jsou mnohem rychlejší než algoritmy přibližného vyhledávání. Velmi vhodné je pak využít algoritmů, jenž pro přesné vyhledávání nepotřebují projít celý text (díky indexování). Díky tomu v mnoha parametrech předčí jiné přístupy k přibližnému vyhledávání [4].

Z výše uvedeného pak vyplývá jistá nevýhoda filtračního přístupu, a sice že je třeba zavést mechanizmus, který zajistí přechod od přesného vyhledání k přibližnému. Typicky je jím takzvaný *verifikátor*, nefiltrující algoritmus, jenž ověří, zda se v oblasti vytipované filtrem skutečně nachází výskyt dle zadání dostatečně podobný vzorku. Základem pro dobrý verifikátor je schopnost dobře fungovat na malých textech, protože lze očekávat, že tímto způsobem bude využit mnohokrát v rámci jednoho dotazu.

Pro další informace o konstrukci filtračního algoritmu viz sekce 2.3. Filtry můžeme rozdělit na dvě skupiny: na jednoduché, které jsou efektivní jen pro krátké až střední vzorky, a složité, jejichž použití se vyplatí pro dlouhé až velmi dlouhé vzorky.

Mezi filtry pro krátké vzorky je zajímavým případem filtr, jenž v roce 1991 vylepšili Jokinen, Tarhio a Ukkonen pro Hammingovu vzdálenost [4]. Princip je založen na posuvném okénku v textu, které v konstantním čase přepočte, kolik kterých znaků se v okénku nachází. Pokud počet znaků odpovídá počtu znaků ve vzorku pro dostatečný počet případů, je obsah okénka prověřen verifikátorem.

Navarro a Baeza-Yates využili v roce 1996 poněkud tradičnějšího přístupu, jenž rozděluje vzorek na několik podvzorků k tomu, aby získali podvzorek dostatečně malý pro použití bitově paralelního algoritmu (se stejným chybovým poměrem jako pro původní vzorek). O rok později byl algoritmus vylepšen tím, že NFA vyhledával několik podvzorků zároveň. Za další rok přibylo vylepšení v podobě tzv. „hierarchické verifikace“, jenž vzorek hierarchicky dělí na podřetězce a verifikován je vždy rodičovský podřetězec [4]. Výhoda spočívá v tom, že pokud je verifikace neúspěšná, tak není verifikován zbytek vzorku.

Mezi filtry pro dlouhé vzorky je vhodné zmínit Ukkonenův algoritmus z roku 1992, jenž využívá typickou techniku vyhledávání všech q-gramů vzorku [4]. Navíc jde o zobecnění výše zmíněného algoritmu pro krátké vzorky, protože filtr sleduje počet nalezených q-gramů v posuvném okénku textu.

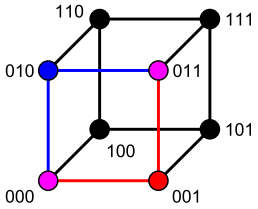
Sutinen a Tarhio byli v roce 1995 první, jejichž algoritmus nesledoval jen správný počet nalezených podvzorků, ale také kontroloval jejich správné pořadí. A také relativní vzdálenost: dva podřetězce vzorku nalézající se v platném výskytu se musí nalézat stejně daleko od sebe jako ve vzorku, či pro editační vzdálenost s rozdílem od této vzdálenosti nejvýše k.

2.2 Metriky a DP

Tato část textu se věnuje využití relevantních algoritmů dynamického programování, vývoji potřeb a historickým souvislostem. Budou také přiblíženy jednotlivé metriky a další termíny, které jsou v práci dále použity.

2.2.1 Hamming

V roce 1950 se Richard Hamming zabýval v [6] detekcí a opravou chyb v přenášených binárních datech. Každý binární řetězec délky n si lze představit jako určitý vrchol n-dimenzionální hyperkrychle. Hodnoty sousedních vrcholů jsou si nejvíce podobné (liší se jen v jediném bitu). Pro jednoduchý příklad viz obrázek 2.2.



Obrázek 2.2: Hyperkrychle pro 3bitová slova a vyznačený příklad vzdálenosti se dvěma chybami (mezi slovy „000“ a „011“). Existují dvě možné posloupnosti chyb (modrá a červená).

S touto představou tak byla elegantně zavedena metrika podobnosti resp. vzdálenosti. *Hammingova vzdálenost* je zde skutečně vzdáleností mezi dvěma řetězci, tedy minimální počet hran, které je nutno projít abychom se dostali z vrcholu jednoho řetězce do vrcholu řetězce druhého. Jedná se o plnohodnotnou *metriku* také v matematickém slova smyslu[7] (je v tomto případě ekvivalentní s *Manhattanskou vzdáleností*[8]). Hranu pak lze chápat jako chybu, jenž mění jeden řetězec v lehce odlišný. *Počet chyb* mezi řetězci je proto jednoznačně definován shodně jako vzdálenost.

Výpočet Hammingovy vzdálenosti není příkladem DP, protože výpočet takto definované vzdálenosti lze realizovat prostým sečtením počtu bitů, kterými se řetězce liší. V praxi se pro zjištění logické nonekvivalence používá operace *XOR* nad zkoumanými vektory[9], následně stačí zjistit počet jedniček výsledku – počet jedniček v binárním vektoru se proto označuje jako *Hammingova váha*.

[7] Splňuje podmínky *nezápornosti, totožnosti, symetrie* a *trojúhelníkovou nerovnost*.

[8] Také nazývaná *součtová vzdálenost*, neboť je definována součtem vzájemných vzdáleností souřadnic, pro všechny dimenze.

[9] Pojmy řetězec symbolů (se $\sigma = 2$), binární vektor, či (počítačové) slovo, jsou v tomto případě zaměnitelné.

Díky této praktické reformulaci se pro srovnání dvou řetězců s větší abecedou používá logicky stejný postup: *Hammingova vzdálenost* je stanovena jako počet nesouhlasných symbolů. Ovšem možná lepším zobecněním je *Leeova vzdálenost*, která byla představena v roce 1958 a navazuje na Hammingovo dílo s ohledem na nebinární data [7]. Definice totiž splňuje představu Manhattanské vzdálenosti pro více dimenzí, na rozdíl od ní ne nad topologií (nekonečné) mřížky, ale toroidu. Ovšem v mnoha aplikacích pozice znaků v abecedě nijak nereflektuje míru podobnosti těchto symbolů.

Leeova i Hammingova vzdálenost se používá v telekomunikaci, proto se Hammingova vzdálenost také někdy nazývá *signálová vzdálenost*.

2.2.2 Levenshtein

V roce 1965 Vladimir Levenshtein přichází opět s tématikou binárních kódů, ovšem opravy chyb nyní zahrnují i přidávání a ubírání znaků [8]. Hammingova metrika srovnává pouze řetězce shodné délky, zatímco Levenshteinova dokáže porovnat řetězce různých délek. Navíc způsob jakým je definována chyba je mnohem bližší způsobu, jak je text tvořen a upravován lidmi, proto se tato metrika velmi často nazývá *editační vzdálenost*.

Každá chyba, respektive opravná operace, je zaměřena na jeden znak:

- *Mismatch*[10] – nahradí znak jiným
- *Insert* – vloží mezi znaky nový (délka řetězce se zvětší)
- *Delete* – smaže určitý znak (délka řetězce se zmenší)

Hodnota *Levenshteinovy vzdálenosti* mezi dvěma řetězci je pak minimální počet těchto operací nutný k přeměně jednoho řetězce v druhý. Tím se praktický výpočet vzdálenosti zesložiťuje na optimalizační problém. Nicméně je zřejmé, že je i zobecněním Hammingovy vzdálenosti (chápanou pro libovolné σ).

Je pravdou, že existují i další metriky odvozené od Levenshteinovy vzdálenosti, čímž vzniká celá tzv. rodina editačních vzdáleností. Z této rodiny je vhodné zmínit metriku, s níž přišel Frederick Damerau v roce 1964. Damerau zkoumal lidské chyby v psaní slov a identifikoval 4 druhy chyb, jenž podle experimentu tvoří 95% lidských chyb [9]. *Damerauova vzdálenost* je tedy definována pomocí těchto 4 operací (protože první tři druhy chyb jsou shodné s Levenshteinovými, je také metrika někdy nazývána Damerau–Levenshteinova). Operací navíc je *transpozice*, čili prohození, dvou sousedních znaků. Díky tomu je metrika nesmírně užitečná pro aplikaci na uživatelské vstupy, avšak značně komplikuje realizaci. Významnou změnou je z teoretického pohledu především to, že pojem jedné chyby se nyní nemusí vztahovat jen k jednomu znaku (ačkoliv každý znak se může účastnit nejvýše jedné operace).

[10] Případně *Substitute*, nebo *Replace*.

2.2.3 Wagner-Fischer

V roce 1974 představili R. Wagner a M. Fisher algoritmus, který řeší optimalizační problém výpočtu editační vzdálenosti pomocí dynamického programování. Avšak možná není spravedlivé uvádět jen jejich jména. Princip řešení tohoto problému je totiž typickým příkladem tzv. vícenásobného objevu. Také jiní vědci jej na přelomu šedesátých a sedmdesátých let objevili, i když oblasti jejich výzkumu se různili. Byli to například: T. Vintsyuk (1968), D. Sankoff (1972), P. Sellers (1974) a mnozí další [4].

Základní myšlenka je totiž typická pro globální metody optimalizace pomocí DP. Jsou totiž vypočteny vzdálenosti mezi všemi prefixy obou řetězců, přičemž se začíná od nejkratších a každé prodloužení je možné spočítat v konstantním čase z optimálních výsledků pro prefix(y) o znak kratší. Výhodou je že tento princip funguje pro každé kritérium vhodné pro tuto dekompozici (tj. každá matematicky korektní metrika).

Avšak pouze Sellers v roce 1980 použil tento princip k přibližnému vyhledávání [4]. Rozdílem oproti výpočtu vzdálenosti vzorku od textu je inicializace matice, jenž nastaví u každé pozice textu nulovou vzdálenost k prefixu vzorku nulové délky (následný průběh algoritmu je také popsán v sekci 2.1.1).

2.2.4 Needleman-Wunsch

V roce 1970 se Saul Needleman a Christian Wunsch stali dalšími objeviteli výše zmíněného algoritmu. Ale cílem jejich výzkumu bylo najít podobnosti v sekvencích aminokyselin dvou proteinů [10]. Tedy ukázat konkrétní místa, kde se sekvence shodují a jakým způsobem je pravděpodobně evoluce změnila.

Tento problém se nazývá *globální zarovnání* a zjednodušeně by se dal popsat jako takový způsob vložení mezer do obou sekvencí, aby byly poté na co nejvíce pozicích shodné. Dá se ukázat[11] že jde o ekvivalentní problém jako je editační vzdálenost:

- *Match / Mismatch* – shodný význam v obou interpretacích
- *Insert* – vložení mezery do první sekvence signalizuje, že během vývoje byl přidán symbol jenž se na této pozici nachází v druhé sekvenci
- *Delete* – vložení mezery do druhé sekvence obdobně signalizuje, že během vývoje byl symbol z první sekvence na této pozici smazán

Avšak algoritmus, dnes běžně nazývaný po svých stvořitelích *Needleman-Wunsch*, je mnohem obecnější, aby mohl lépe vystihovat zákonitosti, jenž byly při zarovnávání biologických sekvencí pozorovány. Toto zobecnění se týká toho, že každá operace[12] může mít jinou cenu. Je tak možné např. upřednostnit

[11] Peter Sellers to v roce 1974 také udělal.

[12] Původní verze z roku 1970 však pevně předpokládala, že vložení mezery bude mít vždy cenu 0. Pro realizované skórování globálního zarovnání viz sekci 4.4.

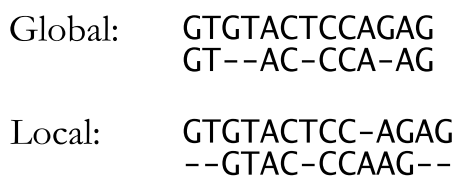
mismatch před vložením mezery, pokud bylo pozorováno, že mutace je pravděpodobnějším vysvětlením než insert resp. delete.

Dokonce i match a mismatch může mít různou úroveň „chyby“ podle toho, které dva symboly jsou porovnávány. Proto lze tyto operace zobecnit na jednu, tedy porovnání, a pomocí matice předurčit výsledky všech možných porovnání. Protože jednotlivé chyby mohou mít různou cenu, přičemž fakticky nemusí jít o chybu, pojem chyba či počet chyb zde zcela ztrácí smysl. Místo toho se používá pojem *skóre*, jenž má prakticky stejný význam jako „cena“, jen s tím rozdílem že ve zobecněné editační vzdálenosti se snažíme cenu chyb minimalizovat, zatímco skóre zarovnání je cílem maximalizovat (chyby jsou zpravidla vyjadřovány záporným skóre, tzv. *penaltou*, zatímco shodné symboly jsou odměněny kladným skóre). Proto se výše zmíněná matice nazývá *skórovací maticí* a číslo na výstupu algoritmu značí podobnost sekvencí, ne jejich vzdálenost.

2.2.5 Smith-Waterman

V roce 1981 pak Temple Smith a Michael Waterman představili variantu Needleman-Wunsch algoritmu pro identifikaci společných molekulárních podsekvencí [11]. Změna spočívá v tom, že tentokrát jde o algoritmus pro *lokální zarovnání*. Obrázek 2.3 ilustruje rozdíl.

Globální zarovnání předpokládá, že obě sekvence jsou téměř stejné a identifikuje rozdíly. Lokální zarovnání naopak identifikuje pouze nejpodobnější podsekvence. Předpokládá se spíš nesoulad velikostí (i obsahu) obou sekvencí a přechod od globálního zarovnání k lokálnímu si lze proto představit podobně jako přechod od algoritmu pro výpočet editační vzdálenosti k Sellersovu algoritmu pro vyhledávání v editační vzdálenosti. Algoritmus je běžně nazývaný *Smith-Waterman* a velice často se využívá právě k vyhledávání určitého motivu ve větší sekvenci.



Obrázek 2.3: Ilustrace rozdílu v obou druzích zarovnání

Z nedávného průzkumu [12] však vyplývá, že není jeden algoritmus vhodnější než druhý (pro molekulární biologii). Za jistých podmínek produkuje kvalitnější zarovnání algoritmus pro lokální zarovnání, ale jindy má kvalitnější výsledky globální zarovnání.

2.3 Filtrační algoritmy a indexy

Většina výše zmíněných algoritmů pro přibližné vyhledávání je koncipována jako *on-line*, kdy může být předzpracován vzorek, nikoliv však text. Přestože existují velmi rychlé algoritmy tohoto typu, pro opravdu velké texty žádný z nich nepřináší uspokojivý výkon [13]. Zaměřím se proto na algoritmy dovolující předzpracování textu a tím i rychlejší vyhledávání. Náš příklad s vyhledáváním v DNA je velmi dobrou ukázkou dlouhého textu, jenž je často prohledáván a indexace se proto vyplatí.

Jak bylo řečeno v úvodu, předpokládá se prohledávání komprimovaných souborů a pro použití klasických algoritmů je proto omezující skutečnost, že se soubor musí celý dekomprimovat. Ve světle těchto omezení tedy navrhuji použít filtrační algoritmus, jenž je schopen využít výhod indexu a pro svou funkci dekomprimuje jen malé části textu (v jeden okamžik typicky zlomek velikosti vzorku). Navíc filtrační algoritmy jsou jako jediné schopny *sublineárního* zpracování (v součtu tedy může být dekomprimována jen část textu, v závislosti na nastavení relativně malá). Pro lepší zobecnění budu dále používat pojem *extrakce* textu, jenž jako operace indexu může znamenat dekomprimování požadované části původního textu.

Existují sice indexy pro přibližné vyhledávání v indexovaném textu, ovšem nejsou tak rychlé a nepřiměřeně navyšují velikost indexového souboru. Indexovací metody pro přesné vyhledávání mají za sebou mnohem delší vývoj a problém, který řeší je o poznání jednodušší. Zaměřím se proto dále na filtrační algoritmy využívající index pro přesné vyhledávání.

Cílem této práce není studium indexovacích metod, nicméně dal jsem si za cíl navrhnout vyhledávání tak, aby bylo možné využít různých indexů. Jednotlivé indexy je možné rozdělit do čtyř hlavních druhů (dle používané datové struktury) – *Suffix tree* a *Suffix array* umožňují vyhledání každého podřetězce textu, zatímco *q-grams* a *q-samples* vyhledávají podřetězce[13] délky nejvýše q [13]. Jak bude později zřejmé, vhodným výběrem (či nastavením) filtračního algoritmu lze vyhovět i tomuto omezení délky při vyhledávání libovolně dlouhého vzorku.

Během vývoje filtračních algoritmů pro různé metody indexování lze vypozorovat, že k významně rozdílným algoritmům bylo dospěno při různém předpokladu toho, kde se chyby vyskytují – buď ve vzorku, nebo v textu [13]. Vzhledem k definicím ze sekce 1.1 budu dále uvažovat jen ten pohled, kde se chyby nacházejí ve vzorku (α je definováno vzhledem k délce vzorku).

2.3.1 Generování okolí

Tento přístup je poměrně přímočarý a opírá se o podobnost výskytů se vzorkem. Nehledě na text je počet řetězců jenž mohou být výskytem konečný. Mohou mít jen určité délky a také jsou počtem tolerovaných chyb omezeny

[13] Metody typu q-samples neindexují všechny q-gramy.

možnosti jejich obsahu. Množina všech těchto řetězců, které mají od vzorku vzdálenost k nebo menší, se nazývá *okolí* vzorku (někdy též k-okolí).

Základní myšlenkou je tedy přesné vyhledávání všech řetězců z k-okolí vzorku. V této variantě se vlastně nejedná o filtrační algoritmus (nicméně výhodou je jednoduchost). Sofistikovanější metody tohoto druhu používají různé metody ořezávání okolí – například pokud není v textu nalezen řetězec z okolí, jenž je zároveň prefixem jiných řetězců z okolí, pak není nutné tyto řetězce vyhledávat. Další vylepšení pak spočívají metodách jak omezit počet redundantních úkonů.

Zásadní slabinou tohoto přístupu je fakt, že okolí vzorku může být velmi velké. Pro podrobnější analýzu viz sekce 3.4. V zásadě roste počet řetězců v okolí exponenciálně s k. Proto generování okolí dobře funguje pro malé m, nízké σ a také nízká k. Avšak tento přístup nemá jen nevýhody – jak vyplyne ze sekce 3.3, pro extrémně krátké vzorky by měl být algoritmus tohoto typu nejlepší.

2.3.2 Filtr pro přesné vyhledávání

Tento přístup se také nazývá *„Partitioning into Exact Search“* [13] a je založen na principu popsaném v sekci 3.1. Hlavní myšlenka je založena na tom, že v každém přibližném výskytu se nachází alespoň jedena část vzorku přesně. Algoritmus proto vyhledává různé podřetězce vzorku, přičemž využívá rychlosti přesného indexovaného vyhledání. Oblasti textu kolem těchto nálezů jsou považovány za *potenciální výskyty* vzorku. Jsou proto extrahovány a prohledány klasickým on-line algoritmem, který určí, zda se jedná o *platný výskyt* vzorku. Právě tento postup je znám jako *filtrace* (viz také sekce 2.1.4).

Typicky je vzorek pravidelně rozdělen na tzv. *podvzorky*, jenž se délkou liší nejvýše o 1. Počet podvzorků je volen tak, aby v případě žádného platného výskytu nemohly být chybami poškozeny všechny podvzorky. Není však zřejmé jaký vztah pro počet podvzorků vzhledem k požadované přesnosti vyhledávání je nejlepší a různí autoři doporučovali protichůdná řešení [13]. S jistotou lze říci, že větší počet podvzorků znamená mnohem více (výsledků) vyhledávání, avšak lze toho využít pro spuštění mnohem méně verifikací, jenž jsou jinak nejnáročnější operací filtrace.

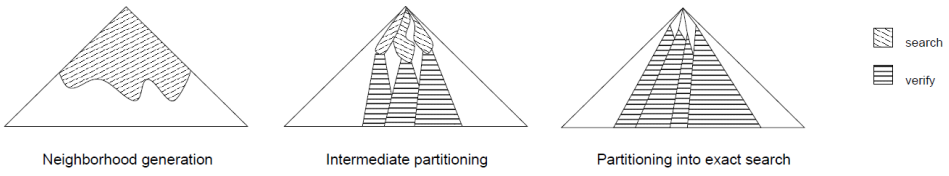
Po rozdělení vzorku jsou všechny podvzorky vyhledávány a všechny jejich výskyty kontrolovány. Při kontrole může být využito faktu, že pořadí podvzorků v potenciálním výskytu musí být zachováno. Dále relativní pozice výskytů podvzorků se mohou lišit oproti přímému navazování nejvýše o k (pro editační vzdálenost). Tyto kontroly mohou zvýšit výkon filtrace, neboť v případě neúspěchu není třeba provádět verifikaci. Samy však mohou být příliš náročnou operací.

2.3.3 Filtr pro vyhledávání s chybami

Tento přístup se také nazývá *„Intermediate Partitioning“* [13] a v zásadě se jedná o spojení obou předchozích. Na rozdíl od filtru pro přesné vyhledávání je vzorek rozdělen na méně podvzorků. Proto nemůže být zaručeno, že ve validním výskytu vzorku se bude minimálně jeden podvzorek vyskytovat bez chyb. Pro každý podvzorek je tedy stanoven maximální počet chyb, se kterými bude vyhledáván. Přestože teoreticky může být v oblasti podvzorku soustředěno až k chyb (aby mohl být výskyt vzorku platný), je možné podvzorky vyhledávat s méně chybami, neboť v duchu předchozího algoritmu stačí vyhledat pouze ten podvzorek výskytu, který obsahuje nejméně chyb.

Samotné vyhledávání podvzorku s takto stanoveným maximálním počtem chyb je řešeno zase algoritmem typu *„Neighborhood Generation“*. Změna parametrů (nižší k i délka vyhledávaných řetězců) tomuto algoritmu významně přidává na efektivitě, jak bylo poznamenáno výše. V sekci 3.3 blíže ukáži, proč může být prodloužení podvzorku výhodou oproti filtrům pro přesné vyhledávání. Avšak pokud by byla délka podvzorku i bez tohoto prodloužení dostatečná, pak je tento přístup v nevýhodě, protože jej významně zpomaluje složitost generování přiblížená v sekci 3.4.

Jak ukazuje obrázek 2.4 jde o kompromis mezi výše uvedenými přístupy, které lze chápat jako extrémní. Experimenty v [14] pak ukazují, že oba přístupy předčí. Stejný zdroj pak uvádí, že optimální počet podvzorků je asymptoticky $\Theta\left(\frac{|P|}{\log_\sigma |T|}\right)$. Ačkoli přesný vztah není znám, také jiní autoři algoritmů tohoto typu objevili závislost odpovídající této.



Obrázek 2.4: Ilustrace poměru času stráveného prohledáváním a verifikací a rozdíly mezi hlavními přístupy přibližného vyhledávání nad indexy [13]

Obecně je tento přístup využíván pro lepší výkon pro vyšší α. Maximální mez poměru chyb pro efektivnost filtračních algoritmů je velmi známá horní mez (např. [4]) uvedená vztahem 2.1. Filtrace se přestává vyplácet, pokud je postupně extrahován celý text. Vzhledem k tomu, že verifikované oblasti se mohou překrývat, nemusí být extrahován doslova celý obsah, aby bylo zpracováno stejné množství dat. Pokud je povolený poměr chyby příliš vysoký, snadno se stane, že přibližné výskyty pokrývají celý text a ani sebelepší fil-

tační algoritmus se postupnému zpracování celého textu nevyhne.

$$\alpha < 1 - \frac{e}{\sqrt{\sigma}} \tag{2.1}$$

Uvedená mez je poměrně pesimistický odhad a v praxi je možno Eulerovo číslo e nahradit jedničkou a počítat s výrazem $\alpha < 1 - \frac{1}{\sqrt{\sigma}}$ [13]. Pro náš příklad s prohledáváním DNA je tedy vhodné klást dotazy s chybovým poměrem nejvýše do 50%. Dle mého názoru pro tuto aplikaci není už tak vysoká chybovost využitelná a tedy filtrační algoritmus tohoto typu může být použit pro tyto účely.

2.4 BLAST

Pro řešení motivační úlohy (v sekci 1.1) se v praxi nejčastěji používá některý z programů BLASTu. Jedná se o specializované řešení pro přibližné prohledávání proteinových řetězců či DNA sekvencí.

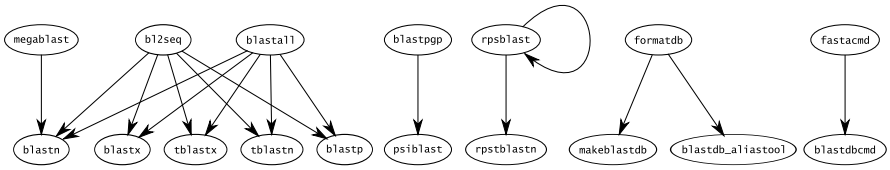
2.4.1 Verze BLASTu

Existuje mnoho BLASTů, já se však zaměřím na originální *NCBI-BLAST*, jehož autorem je Altschul a kolektiv[14] [15]. Algoritmus byl navržen v roce 1990 a vyvíjen v americkém Národním centru pro biologické informace (NCBI), jenž jej umožňuje volně využít. Dále je například známá verze *WU-BLAST*, vytvořená na Washingtonově univerzitě. V současnosti je však vyvíjen společností Advanced Biocomputing pod názvem *AB-BLAST* a není již zdarma pro akademické a nevýdělečné účely [16][15]. Dále existuje řada modifikací jednotlivých programů (NCBI-BLASTu) pro paralelní zpracovávání (nejčastěji akcelerovaných na GPU).

Dále je vhodné zdůraznit, že nástroj BLAST je vlastně kolekcí mnoha různých algoritmů. Obsahuje různé programy dále specializované na určitou abecedu, pro různé délky sekvencí a různé úrovně přesnosti vyhledávání atp. Každý z těchto vyhledávacích programů je možno dále parametrizovat. Vzhledem k motivaci (vyhledávání v DNA) se dále zaměřím na program známý jako *blastn* z nástrojů NCBI-BLAST. Programem „BLAST“ budu tedy pro zjednodušení myslet právě tento program, v jeho implicitním nastavení.

Původní sada nástrojů NCBI-BLAST byla implementována v jazyce C, v současné verzi jsou však programy přepsány do C++[16] a sada je proto označována jako *BLAST+*. Starší verze v jazyce C pak bývá označena jako *Legacy BLAST*. Rozdíl je především v architektuře sady (změny jsou vyobrazeny na obrázku 2.5) a změnách výchozího nastavení. BLAST+ je dále vyvíjen a proto nabízí lepší výkon a více možností (podrobně v [17]).

[14] Stephen Altschul, Warren Gish, Webb Miller, Eugene Myers a David J. Lipman.
[15] Na citované stránce je možné si stáhnout starou verzi WU-BLASTu.
[16] Pro specifické účely jsou součástí i PERL skripty.



Obrázek 2.5: Rozdíly mezi nástroji Legacy BLAST (nahoře) a BLAST+ (dole). Šipky naznačují, které aplikace v nové verzi implementují algoritmy z aplikací staré verze BLASTu [18]

Také se budu zabývat pouze tzv. „Standalone BLAST“, kdy jsou nástroje distribuovány jako nezávislé spustitelné aplikace. Existuje řada alternativ, většinou využívající servery spravující databáze, či webové aplikace poskytující grafické rozhraní [19]. Algoritmy se však neliší.

2.4.2 Vlastnosti BLASTu

Zkratka *BLAST* znamená „Basic Local Alignment Search Tool“ a odkazuje na to, že program vyhledává podobné oblasti dvou sekvencí, spíše než oblasti v sekvenci textu podobné sekvenci vzorku. Přesněji řečeno je BLAST heuristickou metodou pro nalezení nejlepšího lokálního zarovnání. Nezaručuje tedy nalezení optima jako algoritmus Smith–Waterman (viz sekce 2.2.5). Navíc je koncipován jako ztrátový filtr, nalezen je proto jen určitý podíl skutečných výskytů. Na druhou stranu je nespornou výhodou tohoto přístupu rychlost.

Není také zaručeno, jak velká část vzorku bude nalezena, což by v mnoha aplikacích bylo nevýhodné – např. pokud aplikace očekává na výstupu výskyty celého vzorku s nejvýše k rozdíly, musí z výstupu vyhledávání odfiltrovat výskyty, jenž tuto podmínku nesplňují. Reportovanou 100% podobnost může mít i nález jen části vzorku, navíc i mnohonásobně menší délky než zadaný vzorek (za předpokladu, že jde o dostatečně vzácný podřetězec, viz podmínka v následujícím odstavci).

Počet výsledků vyhledávání lze ovlivnit hraniční mírou e-value (podrobně viz sekce 3.3). Ve stručnosti tato veličina představuje, kolikrát by byl daný vzorek pozorován v odpovídající databázi (tedy kolik výskytů lze očekávat v pseudonáhodné DNA). Tento parametr umožňuje ignorovat velmi časté výskyty. Reportovány jsou jen výskyty, kde je hodnota e-value odhadována menší než hraniční (implicitně nastavena na 10) [20].

Ovšem také reportování výskytů splňujících výše uvedenou podmínku je omezeno. Počet výsledků vyhledávání je omezen (ve výchozím nastavení) na 500, respektive počet zarovnání na 250, a ostatní výsledky se zapomínají[17].

Dalším významným rysem je, že BLAST nemá nastavení omezující vyhledávání na určitý počet chyb (ať ve smyslu Hammingovy vzdálenosti, popř.

[17]Zdroje: `blastn -help`, `blastall`.

editační vzdálenosti, nebo dle skóre). To považuji za nevýhodu a dle množství řešení, jenž zpracovávají výstup BLASTu za účelem odfiltrování příliš degenerovaných výskytů, nejsem jediný [18]. V BLAST+ je již integrován takový filtr, jenž umožňuje odfiltrovat z výsledků výskyty s nízkým poměrem shodujících se znaků.

Je možné upravit skóre jednotlivých operací (viz sekce 2.2.4) konkrétně *match*, *mismatch*, *open gap* a *extend gap* (také viz sekce 4.4). Na výběr jsou předem dány určité hodnoty, jenž jsou vzájemně v určitém poměru. Daný poměr (resp. skóre operací) je doporučen pro sekvence zachovávající 99%, 95%, nebo 75% bází (podporované kombinace hodnot jsou uvedeny v [18]).

Při vyhledávání jsou nejednoznačné nukleotidy, tedy znaky '`N`', '`M`', '`W`', apod. (podrobněji viz [21]) vždy považovány za neshodné s jakýmkoli znakem [19]. Avšak časté výskyty takovýchto nejednoznačných bází (každých 10bp) způsobí úplné selhání vyhledávacího algoritmu.

BLAST také považuje vždy za neshodující se celé určité podřetězce. Jedná se o tzv. sekvence s „nízkou složitostí“. Příkladem takové sekvence může být „`AAATAAAAAAAATAAAAAAT`“. Takové podřetězce produkují mnoho výskytů, avšak ve většině případů se v DNA nejedná o sdílenou informaci [20]. Vzorky či výskyty s těmito podřetězci proto mají vždy významně nižší skóre. Nicméně ignorování těchto sekvencí je možné vypnout.

2.4.3 Realizace BLASTu

Je na místě připomenout že BLAST také funguje na principu filtrace nad indexovaným textem. Prohledávaný text, v určitém formátu (FASTA, viz sekce 5.3.1), musí být nejprve zpracován jedním z nástrojů BLASTu jenž vytvoří index. Tento index se skládá typicky z 8 souborů (případně více pokud by soubory byly příliš velké). Tyto soubory jsou následně vstupem vyhledávacích nástrojů (původní soubor textu již není potřeba).

Prvním krokem při vyhledávání je filtrování vstupu, kdy se ze sekvence vzorku vymaskují podřetězce s *nízkou složitostí*. Jedná se především o často opakované části DNA a o delší řady jedné či podobných kyselin (např. pyrimidiny) [22]. Tyto podřetězce by se mohly dostatečně shodovat s relativně náhodnými částmi textu a uměle by navyšovaly skóre jinak nerelevantního výskytu.

Pro identifikaci takovýchto sekvencí existují různé programy. BLAST k tomuto účelu používá pro DNA algoritmus *DUST* (Morgulis a kolektiv, 2006 [23]) a pro řetězce aminokyselin pak algoritmus *SEG* (Wootton a Federhen, 1996 [24]) [20].

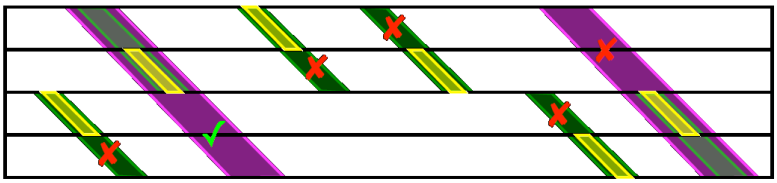
Filtrační algoritmus používá jako podvzorky všechny podřetězce vzorku délky 11 (pro *blastn*). Tato technika je analyzována v sekci 3.2. Následně jsou

[18] Viz např. `https://www.biostars.org/p/2650/`

přesně vyhledávány řetězce z jednotlivých okolí pro všechny tyto podvzorky. V zásadě se tedy jedná o filtr pro vyhledávání s chybami.

Pro samotné vyhledávání všech řetězců z okolí všech podvzorků je použit deterministický konečný automat, který lineárně zpracuje celý vstupní text. Okolí podvzorku je definováno pomocí vážené Levenshteinovy vzdálenosti (blíže viz sekce 3.4). Generováno je však jen tzv. *kondenzované okolí*, jenž neobsahuje řetězce, které jsou vlastním prefixem jiného řetězce z této množiny [25].

Nálezy překrývajících se výskytů se sloučí do jednoho, jenž je „seed“ pro algoritmus *Seed&Extend* [25]. Seed je označení pro část textu, jenž může být také součástí výskytu vzorku. Ověřována je pak jen oblast textu kolem seedu, nikoli celý text. Seedy vhodné délky jsou proto následně verifikovány tzv. „extend“ verifikátorem. Extend provádí iterativní verifikaci střídavě na jednu a poté na druhou stranu, přičemž je vždy zdvojnásobena délka verifikace (proces je znázorněn na obrázku 2.6). Tyto „teleskopické“ verifikace mají za cíl předejít ověřování celé oblasti délky vzorku v případě nevalidního výskytu. Verifikace je definována jen dle vážené Hammingovy vzdálenosti a je zastavena pokud skóre příliš poklesne [25].



Obrázek 2.6: Ilustrace iterativní Extend fáze. Obdélník představuje matici DP kde text je vodorovně a vzorek svisle. Seedy jsou označeny žlutě, jen jeden však byl ověřen jako součást validního výskytu vzorku. [25]

BLAST má díky svému heuristickému přístupu jednodušší filtr než běžné filtrační algoritmy založené na *q-gram* nebo *pigeonhole* principu. Používá také techniku zpracování čtyř znaků DNA řetězce najednou (v jednom bajtu) [26].

Na závěr je vhodné poznamenat, že na výstupu jsou vypsány pozice daných částí textu a vzorku, dále samotné tyto podřetězce a vyznačeno jejich vzájemné zarovnání. Zobrazeny jsou ke každému výskytu také hodnoty jednotlivých metrik podobnosti.

Kapitola 3

Analýza a návrh

3.1 Pigeonhole Principle

Tento matematický princip je poměrně jednoduchý, avšak objevení (či znovuobjevení) této myšlenky přispělo k vývoji nejedné vědecké disciplíny. Poprvé jej zformalizoval matematik Peter Gustav Lejeune Dirichlet v roce 1834 [27]. Myšlenku však pojmenoval „zásuvkový princip“ a pod tímto pojmenováním je proto známa v mnoha jazycích (anglicky *Drawer principle*).

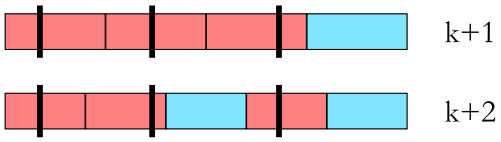
Populární vysvětlení principu používá holuby, jenž se snaží narychlo ukrýt před bouří. Holubník má několik „budek“ (*pigeonholes*), jejich však méně než holubů. Holubi v panice nemusí obsadit všechny budky, v některých se naopak může tísnit více holubů. Princip však praví, že v každém případě nalezneme alespoň jednu budku, ve které se budou ukrývat minimálně dva holubi (jednoduše proto, že není budka pro každého).

Tento fakt lze různými způsoby zobecnit, či stanovit vzorcem pravděpodobnosti jednotlivých jevů. Tímto principem lze například vysvětlit *narozeninový paradox*, či snadno dokázat, že v Londýně žijí nejméně dva lidé s přesně stejným počtem vlasů. Já jej využiji jen v souvislosti s přibližným vyhledáváním nad vlastními indexy (tato myšlenka je také pevně spjata s *hashovacími* algoritmy, jenž jsou při realizaci relevantní, ale nebudu se jim dále věnovat).

Z pohledu analytické filosofie není třeba matematických vyjádření, abych v našem případě s komprimovanými texty připomněl jednu důležitou skutečnost. A sice že kvůli pigeonhole principle není možné nalézt kompresní algoritmus, jenž by každý text dokázal beze ztrát zkomprimovat do menšího souboru. Počet všech textů určité délky je totiž vždy větší než počet různých souborů určité délky, která je menší než zmíněná délka původního textu.

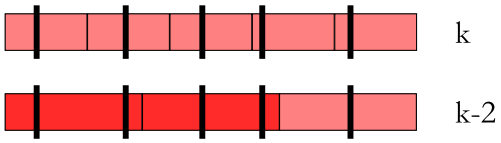
V realizaci základního filtračního algoritmu, jakým je např. „MultiPEX“ [28], se pigeonhole principle uplatňuje při volbě počtu podvzorků. Jedná se o filtr typu *partitioning into exact search* a proto je nutné zajistit, že alespoň jeden podvzorek nebude obsahovat chybu. Algoritmy rodiny „PEX“ proto vzorek dělí na $k+1$ částí. Jiné algoritmy dělí na více částí, čímž je dle popiso-

vaného principu zajištěn větší počet vzorků bez chyb a proto dokáží vyhledat větší část vzorku, jenž je také obsažena v textu (viz také sekce 2.3.2). Tyto rozdílné přístupy jsou vysvětleny obrázkem 3.1.



Obrázek 3.1: Funkce různých filtrů pro přesné vyhledávání: Obdélník představuje vzorek rozdělený na podvzorky. Řekněme, že $k = 3$, pak je vyobrazen nejhorší scénář výskytu – chyby jsou ve výskytu vyznačeny černými značkami.

Pigeonhole principle jsem také využil při realizaci filtrů typu *intermediate partitioning*. Základní přístup spočívá v rozdělení vzorku na právě k částí. Jak ukazuje obrázek 3.2, v nejhorším případě není ve výskytu žádný podvzorek bez chyby. Stačí ale vyhledávat s jedinou chybou. Avšak výkon filterace závisí na délce podvzorku (viz sekce 3.3). Je tedy možné prodloužit délku podvzorku aniž by bylo nutné vyhledávat s více chybami? Ano, z pigeonhole principle vyplývá že podvzorky je možno prodlužovat dokud alespoň jeden podvzorek bude obsahovat i v nejhorším případě nižší počet chyb než ostatní (tato situace je také vyobrazena na obrázku 3.2).



Obrázek 3.2: Funkce různých filtrů pro vyhledávání s chybami: V obou případech je možné vyhledávat jen s jednou chybou, ale ve druhém případě algoritmus vyhledává podvzorky téměř dvojnásobné délky. Zvolené $k = 5$ také ilustruje vhodnost filtrů tohoto typu pro vysoké α.

Při implementaci generalizovaného filtračního algoritmu (viz sekce 3.5), jenž vzorek dělí na $k+s$ podvzorků, jsem proto použil vztah 3.1 pro stanovení minimální hodnoty k, s jakou je nutné vyhledávat podvzorek v indexu. Číslo je označeno jako k_p, aby bylo odlišeno od hodnoty k stanovené pro celý vzorek.

$$k_p = \left\lfloor \frac{k}{k+s} \right\rfloor \tag{3.1}$$

3.2 Q-lemma

Q-lemma, nebo plným názvem *q-gram Lemma*, kterou představili Jokinen a Ukkonen v roce 1991 [26], je důležitou pomůckou zejména pro „q-gram fil-

try“. Tyto filtrační algoritmy většinou pracují nad indexem, jenž je schopen vyhledávat jen řetězce omezené délky (viz sekce 2.3). Tuto délku označuje číslo q. Pomocí přesného vyhledání jsou indexem lokalizovány všechny *q-gramy* vzorku. Každý podřetězec délky q, ovšem typicky není kontrolováno, zda jsou různé úseky vzorku shodné.

$$t = |P| - q + 1 - k \cdot q \tag{3.2}$$

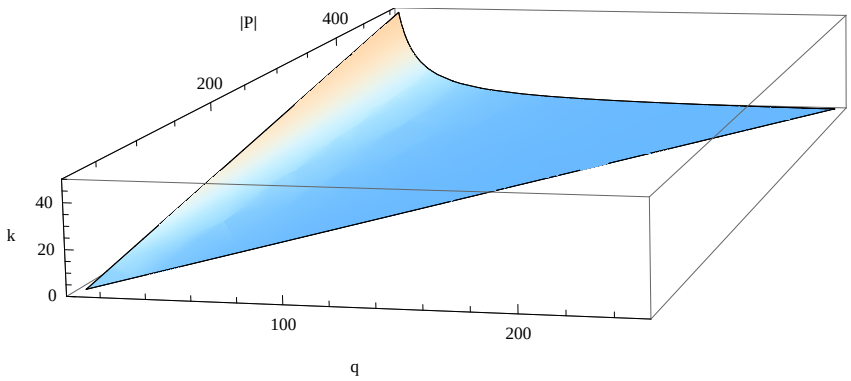
Lemma říká, jaký nejmenší počet q-gramů vzorku musí obsahovat validní výskyt (v rovnici 3.2 je tento počet označen jako t). Vzorek je možno rozdělit na $|P| - q + 1$ q-gramů, jenž jsou dále považovány za podvzorky. Protože q-gramy se překrývají (vyjma okrajových částí se na každém místě překrývá q podvzorků), každá chyba tedy ovlivní q podvzorků. Nejhůře tedy $k \cdot q$ ze všech q-gramů nebude nalezeno. Aby filtr nebyl ztrátový, musí být $t \geq 1$. Z q-lemmy pak vyplývá, že pro parametry vyhledávání musejí být splněny tyto podmínky:

$$|P| \geq q + k \cdot q \tag{3.3}$$

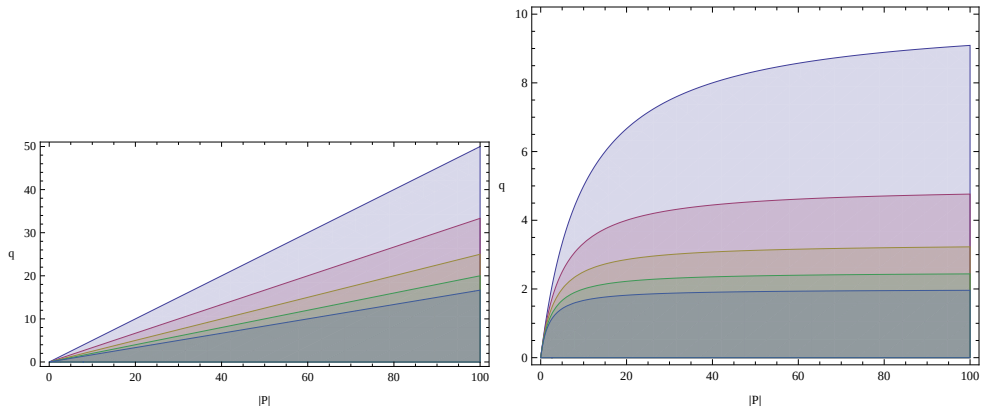
$$k \leq \frac{|P| - q}{q} \tag{3.4}$$

$$q \leq \frac{|P|}{k + 1} \tag{3.5}$$

Na grafu na obrázku 3.3 jsou vyobrazeny kombinace parametrů splňující výše uvedené podmínky. Jak je vidět, prostor validních parametrů je poměrně malý. S rostoucím k (pro nejdelší vzorek je zobrazeno do $\alpha = 10\%$) velmi rychle klesá maximální q. Např. pro $\alpha = 35\%$ je možné vyhledávat nejvýše po dvou znacích, což je problematické (viz sekce 3.3).



Obrázek 3.3: Prostor pod modrou plochou splňuje požadavky pro *partitioning into exact search*



Obrázek 3.4: Prostor pod křivkou je validní nastavení pro danou úroveň povolené chybovosti: vlevo pro k 1–5, vpravo pro α 10%–50% (pro nižší hodnoty je prostor větší).

Z grafů na obrázku 3.4 je možné odečíst nastavení pro vzorky do délky 100 znaků. Pro lepší představu jsou všechny grafy vykresleny spojitě, je však nutno upozornit, že sledovaný prostor je diskrétní.

Existují některá vylepšení výše popsaného algoritmu [26]. Například vyhledávání s chybami slouží k překonání podmínky $t \geq 1$ (za cenu větší složitosti vyhledávání jednoho q-gramu). Velmi vhodná je technika „samplování“, kdy jsou vyhledávány jen některé q-gramy. S úspěchem byla tato technika použita např. v [29]. Extrémním případem je pak použití pouze disjunktních q-gramů. Tomuto případu odpovídá dělení vzorku dle *pigeonhole principle* jak bylo popsáno v sekci 3.1. Z důvodu nízkého počtu vyhledávacích dotazů navrhuji toto schéma dělení (q-gramy dále nebudou zmiňovány).

3.3 E-value

Dosud jsem v práci nejednou naznačil, že délka podvzorku hraje důležitou roli, nyní vysvětlím jakou. Filtrační algoritmus pro každé přibližné vyhledání použije sadu dotazů pro přesné vyhledání. Každý dotaz pak pomocí indexu lokalizuje všechny výskyty požadovaného podvzorku v textu. Filtr následně musí všechny tyto výskyty (pro všechny podvzorky) ověřit. Ověření většinou zahrnuje proces verifikace, jde tedy o poměrně drahou operaci.

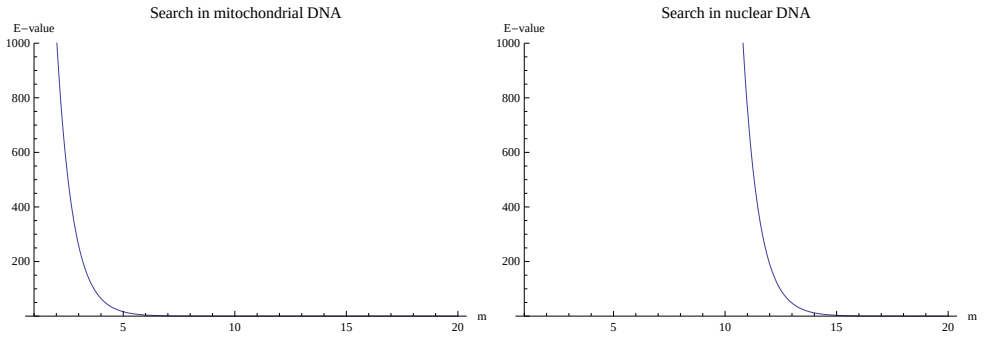
Počet skutečných výskytů podvzorku může být pro rychlost filtračního algoritmu zásadní. Stejný efekt se přirozeně objevuje i v procesu přesného vyhledávání indexem. Ačkoli nemůžeme předem znát počet výskytů určitého řetězce, je možné jej odhadnout. Ve specializovaných aplikacích je možné odhad zpřesnit např. empirickým výzkumem (použito pro algoritmus „PSI-BLAST“ v [30]). Já ale kvůli obecnosti budu předpokládat, že řetězce textu i pod-

vzorků se svými vlastnostmi neliší od náhodných řetězců odpovídající délky nad danou abecedou.

Hodnota E, či též tzv. *e-value*, pak udává, kolik výskytů lze očekávat („Expect“). Při hodnotách menších než jedna, je vhodnější e-value interpretovat jako hodnotu pravděpodobnosti toho, že se hledaný řetězec v textu vyskytuje. V souladu se značením ze sekce 1.1, jsem e-value definoval takto:

$$E = \frac{n - m + 1}{\sigma^m} \tag{3.6}$$

Ze vzorce je patrné že délka (pod)vzorku, značená m má významný vliv. Pro delší podvzorky rychle klesá, avšak pro krátké podvzorky stoupá k hodnotám, jenž naznačují, že ověřen musí být celý text. Na grafech na obrázku 3.5 je vidět, že tato hranice je poměrně ostrá. Při zpracování DNA je pak e-value zvláště vysoká díky dlouhým genomům (n) a malé abecedě. Vliv délky genomu je z grafů patrný[19].

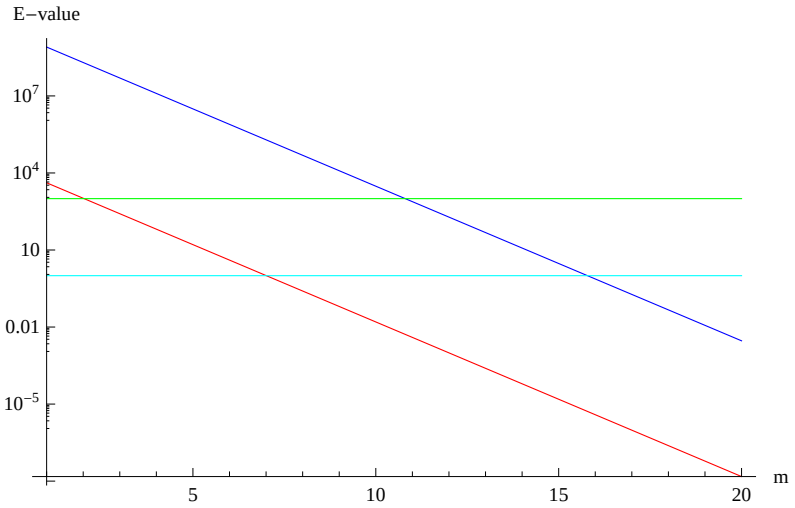


Obrázek 3.5: Volba vhodné délky podvzorků může záviset na velikosti textu (případ vyhledávání v lidské DNA – vlevo mitochondriální, vpravo jaderná)

Pro praktické využití filtračních algoritmů navrhuji řídit nastavení velikosti podvzorků s ohledem na hodnotu E. Například pro $E < 1$ je vhodné zvážit, zda počet podvzorků není zbytečně nízký, zvláště v případě filtru vyhledávání s chybami, protože počet dotazů by mohl výrazně převyšovat počet nálezů. Naopak vysoké E, jehož hraniční hodnota by mohla být stanovena řádově, by mělo varovat před příliš vysokým počtem podvzorků, zvláště pro filtr s přesným vyhledáváním. Rozhodování může být založeno na modelu vyobrazeném na grafu na obrázku 3.6, hodnota E pak dále musí být násobena počtem dotazů – ten sám o sobě může být aspektem rozhodování. Změna filtračního algoritmu (resp. jeho nastavení) však musí být zvážena i z hlediska nastaveného poměru chyb.

Dodatečný algoritmus by před spuštěním filtrace mohl například upravit parametry filtru tak, aby e-value co nejlépe odpovídala doporučené hodnotě,

[19] Mitochondriální část genomu je asi 200000-krát menší než jaderná.



Obrázek 3.6: Logaritmický graf E-value pro mitochondriální (červeně) a jadernou (tm. modře) lidskou DNA. Světle modrá ryska označuje 1 očekávaný výskyt, zelená ryska hranici 1000 výskytů. Mezi průsečíky s těmito ryskami (hodnoty hranic jsou příklad) lze odečíst interval vhodných délek podvzorků.

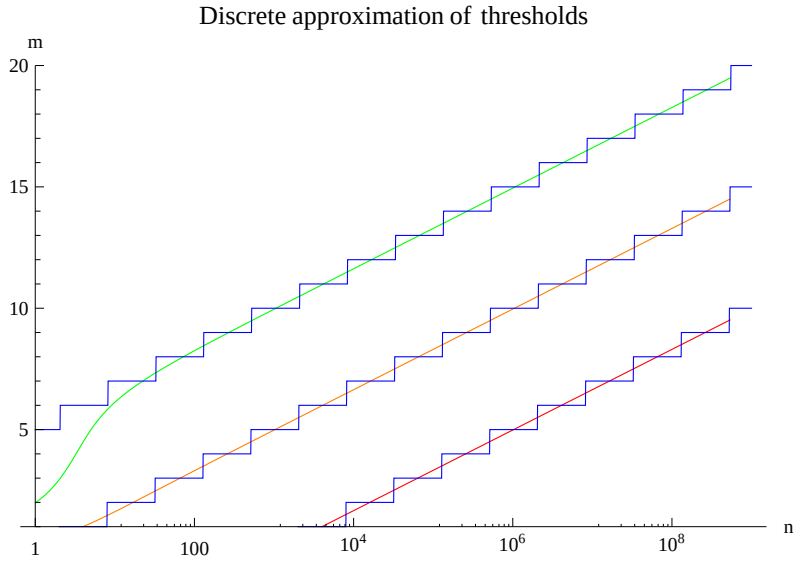
která by nejspíše byla technologicky závislá. Avšak analytické nalezení průsečíků z výše uvedené rovnice je relativně komplikované (bylo by třeba vypočítat Lambertovu „W-funkci“ [31]). Navrhuji proto situaci zjednodušit zanedbáním velikosti podvzorku vůči velikosti textu. Pak lze prahové hodnoty délky podvzorků poměrně jednoduše vypočítat vztahem 3.7 (logaritmus je třeba zaokrouhlit, protože m je počet znaků).

$$m \approx \log_\sigma \left(\frac{|T|}{E} \right) \tag{3.7}$$

Na grafu na obrázku 3.7 je možné srovnat, jak se takto odhadnutá délka podvzorku liší od analytického řešení v oboru reálných čísel. Mimochodem takový vztah pro stanovení dělení vzorku vyhovuje optimálnímu chování dělení, které bylo zmíněno v sekci 2.3.3.

3.4 K-okolí

Jak bylo řečeno v sekci 2.3.1, generování okolí je jedním ze základních způsobů vyhledávání s chybami (pomocí indexu). Je třeba vygenerovat všechny řetězce, jenž mají od vyhledávaného vzdálenost k, nebo menší. Tato množina řetězců se nazývá *k-okolí*. Je na místě poznamenat, že toto k nemusí být shodné s povoleným počtem chyb vzorku, neboť při redukci problému je vyhledáván jen podvzorek a k je také úměrně sníženo (viz sekce 3.1). Přesto může být velikost



Obrázek 3.7: Logaritmický graf ověřující, že navržené zjednodušení výpočtu délky podvzorku (modře) je dostatečně přesné pro prakticky všechny délky textu (do řádu miliard znaků). Zobrazeno pro tyto hodnoty E: 0,001 (zeleně), 1 a 1000 (červeně). Vzhledem k problematice DNA bylo dosazeno $\sigma = 4$.

okolí pro nevhodné nastavení příliš velká. Nyní tedy přiblížím chování velikosti k-okolí pro Hammingovu i Levenshteinovu vzdálenost.

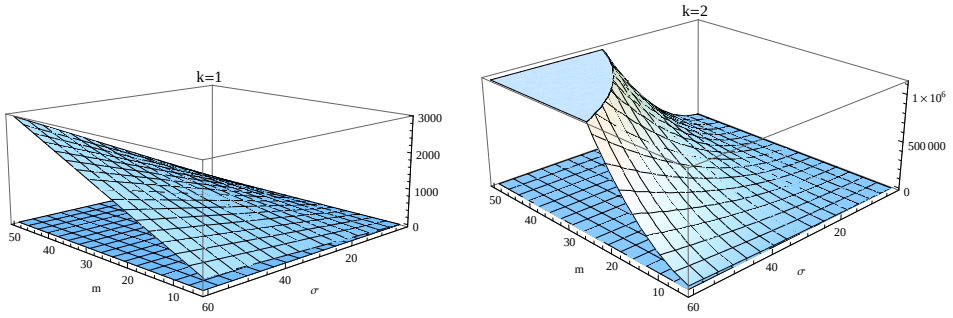
Definovat velikost k-okolí v Hammingově vzdálenosti (označeno H) je poměrně snadné. Zavedu při této příležitosti pojem *k-podokolí* jenž označuje množinu všech řetězců nad danou abecedou, jenž mají vzdálenost od hledaného řetězce právě k. Je zřejmé, že okolí je sjednocením takovýchto disjunktních množin k'-podokolí pro všechna $0 \leq k' \leq k$. Velikost k-podokolí v Hammingově vzdálenosti je označena H_k a její výpočet rovnicí 3.8 lze vysvětlit jako výběr k různých pozic, na nichž je znak nezávisle nahrazen jakýmkoli znakem abecedy kromě totožného.

$$H_k = \frac{m! \cdot (\sigma - 1)^k}{k! \cdot (m - k)!} \tag{3.8}$$

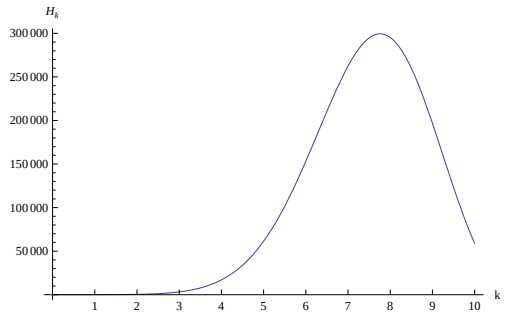
$$H = \sum_{i=0}^{k} H_i \tag{3.9}$$

Z vyobrazení velikosti jednotlivých podokolí na obrázku 3.8 vzhledem k délce podvzorku a velikosti abecedy je vidět, že 1-podokolí je lineárního charakteru, zatímco 2-podokolí je kvadratické. Vzájemné vztahy následujících podokolí jsou analogické. Z toho vyplývá, že vyhledávání s chybami je vhodné omezit na jednu chybu, nebo zajistit, že použitá abeceda, nebo délka pod-

vzorku budou dostatečně malé. Je vhodné zmínit fakt, že v určitých případech velikost k-podokolí klesá – především když se k blíží k m. Takový případ je vyobrazen na grafu na obrázku 3.9



Obrázek 3.8: Velikost podokolí v Hammingově vzdálenosti: vlevo sedlovitá plocha pro $k = 1$ nad rovinou představující $k = 0$, vpravo pro $k = 2$ (oříznuto) nad plochou pro $k = 1$, jenž se v poměru k této jeví jako rovina.



Obrázek 3.9: Velikosti podokolí nejsou nutně monotónní – vyobrazen je případ pro $m = 10$ a $\sigma = 4$.

V případě editační vzdálenosti bohužel situace není tak jednoduchá. Přesné a obecné vyjádření velikosti k-okolí je obtížné. Lze ale použít odhadu 3.10 [25].

$$N = \frac{m! \cdot (2 \cdot \sigma)^k}{k! \cdot (m - k)!} \tag{3.10}$$

Funkci N lze vysvětlit podobně jako rovnici 3.8 pro výpočet H_k, avšak s tím rozdílem že chybou může být: $\sigma - 1$ záměn hodnoty znaky, jedenkrát odstranění znaku, a σ vložení libovolného symbolu abecedy. Celkem tedy existuje $2 \cdot \sigma$ možností jak každou chybu realizovat. Podle vzorce jsou chyby nezávisle realizovány na k různých pozicích. Jedná se tedy o odhad velikosti k-podokolí, velikost k-okolí by bylo možné odhadovat analogicky k rovnici 3.8. Původní autor situaci zjednodušil pravděpodobně na základě předpokladu, že nižší podokolí jsou zanedbatelné.

Tento model však umožňuje vložit nový symbol jen za daný znak – nelze tak vkládat na začátek řetězce. Dalším důsledkem je například nemožnost vložit více znaků za sebe, jedině že by znak kterým jsou vložené symboly proloženy byl smazán. Podle Levenshteinovy vzdálenosti by ale taková situace měla být hodnocena jako jedna editační operace, totiž nahrazení, nikoli dvě.

Proto jsem si zavedl přesnější odhad 3.11, jedná se o velikost 1-podokolí pro editační vzdálenost.

$$E_1 = \sigma \cdot (2 \cdot m + 1) \tag{3.11}$$

Jak je vidět vzorec je velmi jednoduchý a při strojovém hledání optimálního nastavení také rychle vypočitatelný. Při konstrukci tohoto vzorce jsem využil toho, že v případě jedné chyby jsou jednotlivé druhy chyb disjunktní jevy. Pak *Mismatch* je $(\sigma - 1)^k \cdot \binom{m}{k}$, *Delete* jen $\binom{m}{k}$ a *Insert* je $\sigma^k \cdot \binom{m+k}{k}$. Následně se dá ukázat, že po dosazení $k = 1$ je součtem E_1. Mimochodem obdobný vzorec pro Hammingovo 1-podokolí je $m \cdot (\sigma - 1)$.

Podobným způsobem jsem pomocí „IN-EX“[20] vyjádřil i odhad pro E_2. Ale vzhledem k jeho komplikovanosti jej zde nebudu uvádět (nejvýznamnější člen je $\sigma^2 \cdot m^2$). Obecně je dle [13] pro velikost okolí dobrým odhadem i $\mathcal{O}(\sigma^k \cdot m^k)$. Avšak vzhledem k rychlosti růstu okolí předpokládám, že není mnoho důvodů ke generování okolí s $k > 2$.

Při stanovování E_2 jsem se zabýval eliminací neoptimálních kombinací editačních operací. Levenshteinova vzdálenost je založena na minimálním počtu editačních operací, které jsou třeba k transformaci vzorku na jiný řetězec. Každá sekvence operací tedy není validní, respektive produkuje řetězec náležící do jiného podokolí vzorku.

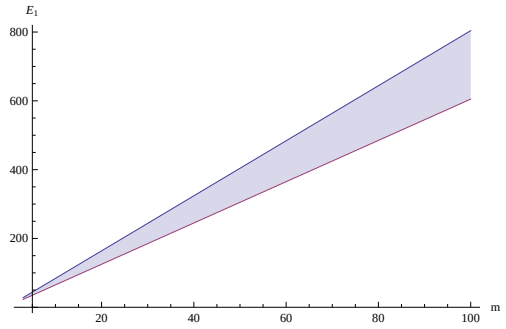
Přesto je třeba připomenout, že vyjádření velikosti podokolí by nemohlo být přesné. Odhadem je i E_1, konkrétně horní mezí. Důvodem je, že velikost k-okolí je pro editační vzdálenost datově citlivá – vliv je znázorněn grafem na obrázku 3.10. Pravděpodobně „nejhorším“ případem je podvzorek, jenž je sekvencí shodných symbolů. Smazání libovolného symbolu se projeví stejně. Rovněž vložení téhož symbolu na libovolnou pozici. Tímto způsobem jsem stanovil naopak spodní mez 1-podokolí:

$$E_1' = 1 + \sigma + 2 \cdot m \cdot (\sigma - 1) \tag{3.12}$$

3.5 Výběr filtračního algoritmu

Místo implementace třech různých algoritmů dle sekce 2.3 s parametrem upravujícím délku podvzorku, jsem se rozhodl situaci zjednodušit a realizovat jeden

[20] Princip inkluze a exkluze, používaný při vyjádření mohutnosti sjednocení nedisjunktních konečných množin.



Obrázek 3.10: Graficky znázorněné rozmezí pro vliv obsahu podvzorku na velikost 1-podokolí v editační vzdálenosti (pro $\sigma = 4$).

obecný algoritmus, taktéž s parametrem upravující počet podvzorků. Zmíněné druhy algoritmů lze zobecnit na filtr pro vyhledávání s chybami, přičemž jsou vyhledávány jen podvzorky s nejmenším nutným počtem chyb (viz sekce 3.1).

Pokud počet podvzorků obecně stanovím jako $k + s$, pak parametrem s lze zároveň provést výběr algoritmu. Pro $s \geq 1$ se bude filtrace vyhledávat bez chyb (dělení odpovídá požadavkům pro „Partitioning into Exact Search“, kdy je vyhledáváno s podvzorků neobsahující chyby). Pokud je nastaveno $s = -(k - 1)$, není vzorek dělen vůbec a chování algoritmu odpovídá „Neighborhood Generation“. Jestliže bude hodnota $1 > s > 1 - k$ jedná se o „Intermediate Partitioning“, což odpovídá tomu, že jde o kombinaci obou předchozích přístupů.

Pomocí vzorců vyjadřujících e-value a velikost k-okolí, je pak možné odhadnout pravděpodobný počet operací provedených pro určitý dotaz na přibližné vyhledání vzorku. Dle návrhu v sekci 3.3 jsem složitost filtrace modeloval jako celkový počet verifikací a počet dotazů na přesné vyhledání. V další práci by mohla být jednotková cena nahrazena cenou odpovídající složitosti těchto operací. Model použitý na grafech odpovídá realizovanému „Seed&Extend“ algoritmu (viz sekce 4.2):

$$(H + E \cdot H) \cdot (k + s) \tag{3.13}$$

Velikost problému je redukována počtem podvzorků. Nerealizovaným zpřesněním je uvažovat celočíselnou hodnotu délky podvzorků, jenž příznivě ovlivňuje e-value a nepříznivě generování okolí. Vzorec by stačilo rozdělit na výpočet pro delší a kratší[21] podvzorky, přičemž počet delších vzorků je roven $|P|$ modulo $(k + s)$. V případě editační vzdálenosti je H nahrazeno N.

Realizovaný algoritmus nedokáže při $s > 1$ využít výhody filtrování dle relativní pozice výskytů, proto na grafu na obrázku 3.11 očekávám zhoršující se tendenci při zvyšování s. Pro metodu „Partitioning into Exact Search“ proto

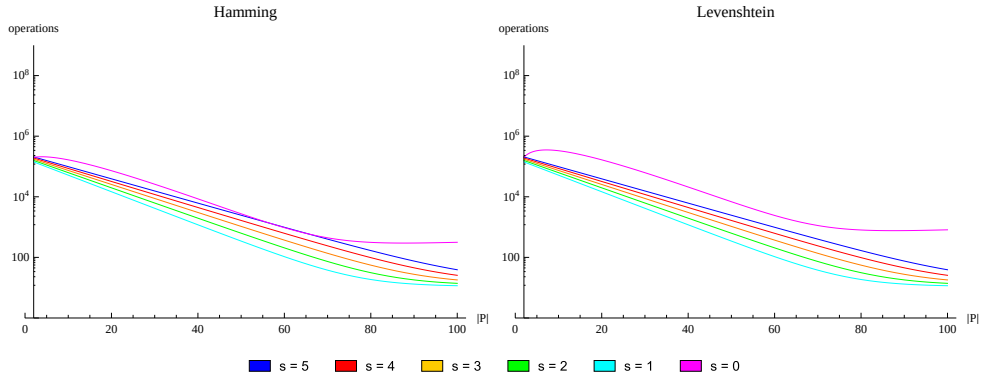
[21] Velikost jednotlivých podvzorků se může lišit nejvýše o 1.

doporučuji používat $s = 1$. Pro metodu „Intermediate Partitioning“ pak na základě domněnky ze sekce 3.4 navrhuji použití takového nastavení, jenž maximalizuje délku podvzorku (kvůli předpokladu že prohledávaný text je velký), při zachování nejmenšího nutného okolí ke generování (tedy vyhledávání s 1 chybou). K tomu je třeba použít nejmenší s splňující podmínku 3.14. Podle této hraniční hodnoty byly také odděleny grafy na obrázcích 3.12 a 3.13.

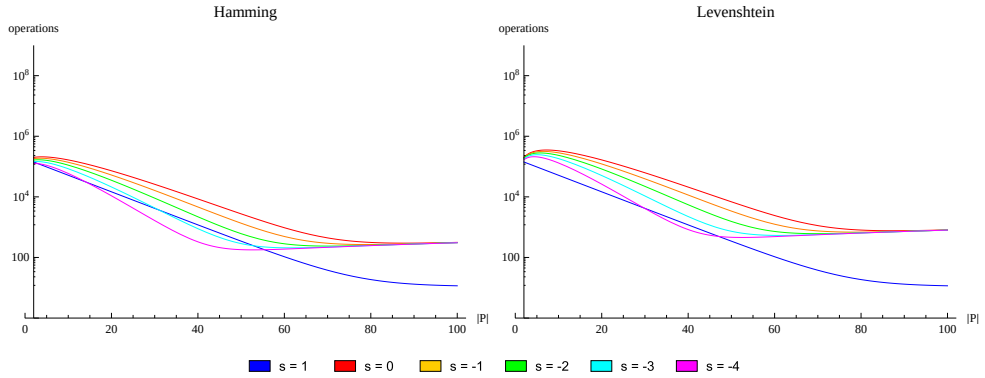
$$s > -\frac{k}{2} \tag{3.14}$$

Níže uvedené grafy představují srovnání filtračních algoritmů s různým parametrem s dle modelu 3.13. Prohledávána byla mitochondriální DNA, jenž je relativně krátká (s předpokladem $\sigma = 4$). Vyobrazená je závislost na délce vzorku, vždy pak bylo vyhledáváno s deseti chybami (tedy $k = 10$). Grafy tedy odpovídají klesajícímu chybovému poměru ($\alpha \geq 10\%$). Vlevo je vyobrazeno vyhledávání v Hammingově vzdálenosti a vpravo v Levenshteinově. Všechny grafy jsou v logaritmickém měřítku.

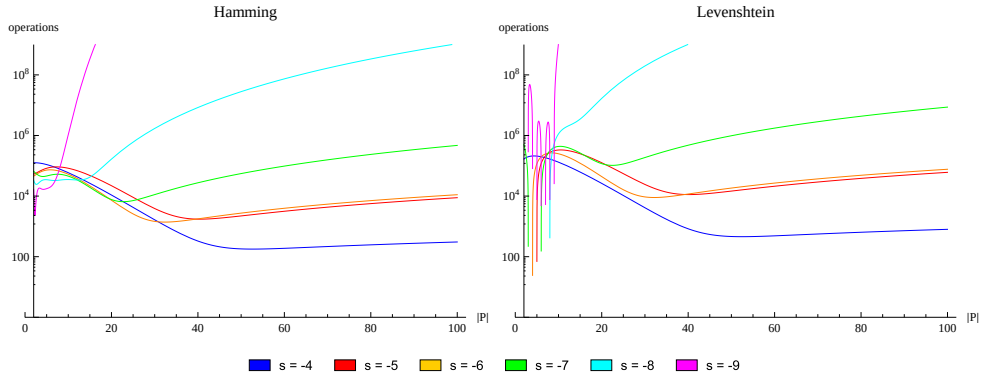
Na následujících grafech ukáži, jak se situace může změnit. Prohledávání bude modelováno nad celým lidským genomem, viz obrázek 3.14. Ukáži také, jaké očekávám chování pro konstantní α při různých délkách vzorku – grafy na obrázcích 3.16 a 3.15. Porovnána bude filtrace s vyhledáváním bez chyb ($s = 1$), oproti vyhledávání s chybou, kde s bude dynamicky měněno a vybráno výše navrženým postupem tak, aby délka podvzorků byla maximální a nebylo nutné vyhledávat podvzorky s více než jednou chybou. Na závěr následuje případ změny velikosti σ při prohledávání textu v přirozeném jazyce (na základě změření abecedy česky psaného textu, s rozlišením velikosti písmen, jsem použil $\sigma = 100$), viz graf na obrázku 3.17. Kde není uvedeno jinak, je vyhledáváno s deseti chybami v editační vzdálenosti. Upozorňuji, že výsledky na grafech jsou jen odhady.



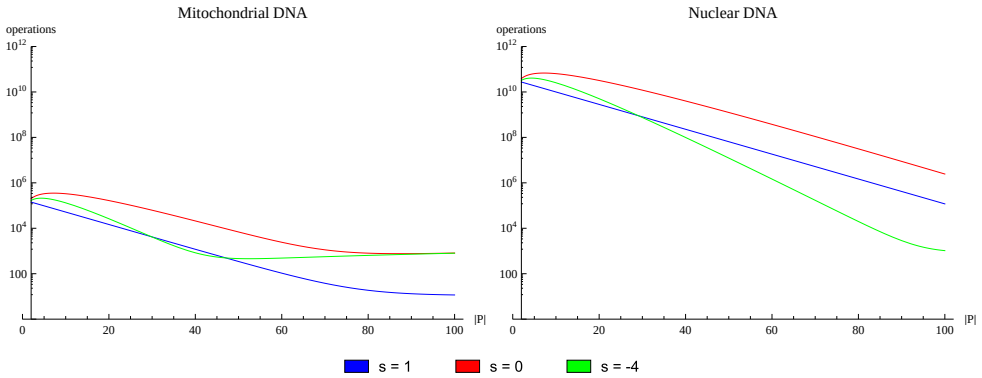
Obrázek 3.11: Snižující se s má pro „Seed&Extend“ lepší výsledky až do přechodu na vyhledávání s chybou (fialově).



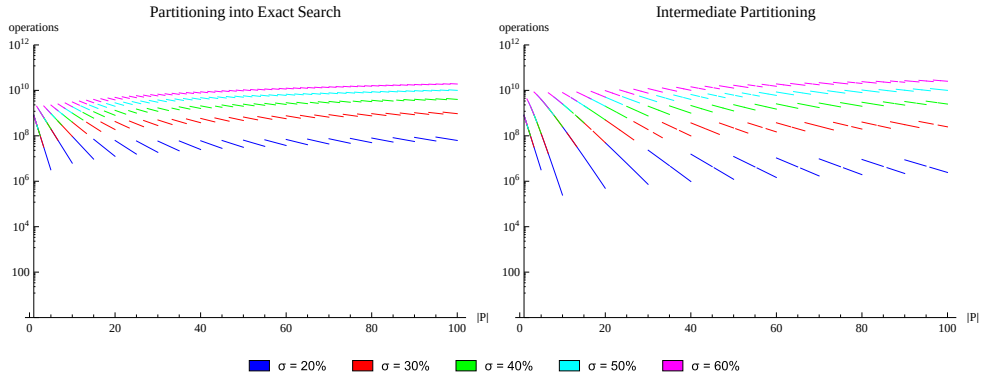
Obrázek 3.12: Nejnižší s pro vyhledání s chybou má v jistém rozsahu (pro Hamingovu vzdálenost při $|P|$ mezi 15–55) lepší výsledky než nejlepší vyhledávání bez chyb (modře).



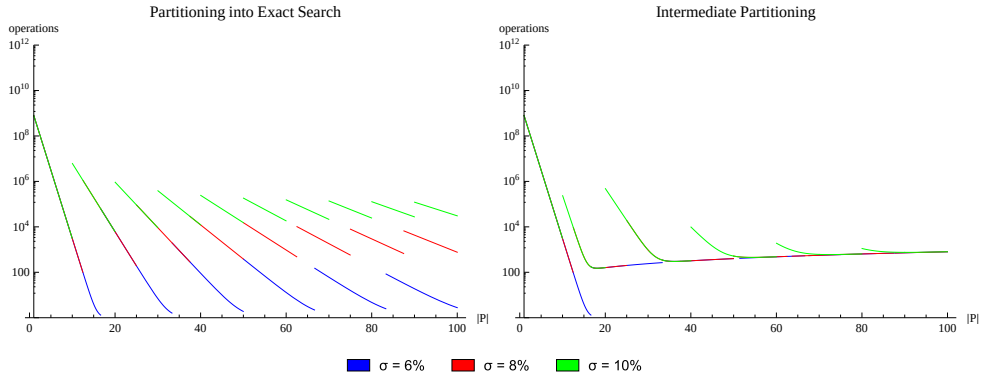
Obrázek 3.13: Zdá se, že pro krátké vzorky by mohlo jít jen vzácně o rychlejší postup než pro nejrychlejší vyhledávání s 1 chybou (modře). Failová linka odpovídá přístupu generování celého okolí.



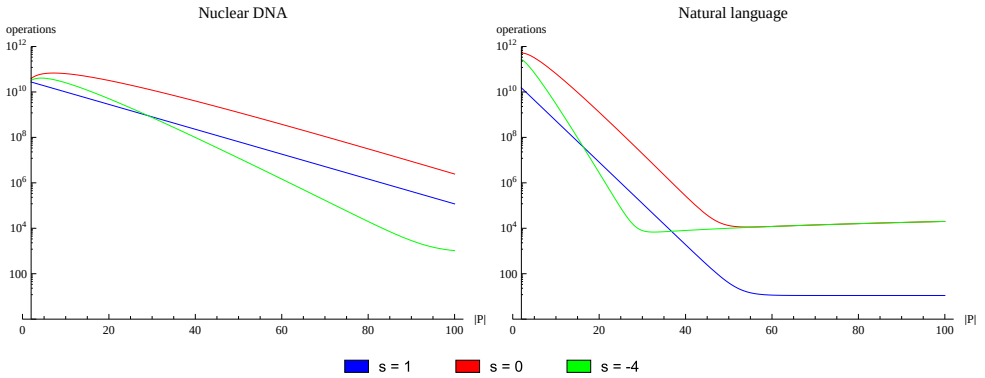
Obrázek 3.14: Na těchto grafech je vidět že vyhledávání závisí na délce textu. Ačkoli se křivky protínají stejným způsobem, pro dlouhé texty je vidět že dobré nastavení filtru s vyhledáváním s chybami může být i o několik řádů rychlejší než filtr s vyhledáváním bez chyb. Rozsah na kterém výhoda platí je navíc poměrně široký (30–140bp).



Obrázek 3.15: Pro vysoký chybový poměr vítězí dle modelu filtrace s vyhledáváním s 1 chybou, nad filtrem s vyhledáváním bez chyby, ze zkoumaných situací pro $\alpha < 50\%$ (pro větší poměry není výsledek zřejmý).



Obrázek 3.16: Pro nižší chybový poměr je do $\alpha \geq 8\%$ výhodnější filtr s vyhledáváním s 1 chybou, pro $\alpha \leq 6\%$ se zdá lepší filtrace s vyhledáváním bez chyby, protože není zatížena generováním okolí. V hraničním případě lze říci, že filtrace s vyhledáváním bez chyby nemá tak vyrovnané výsledky (je více závislá na $|P|$).



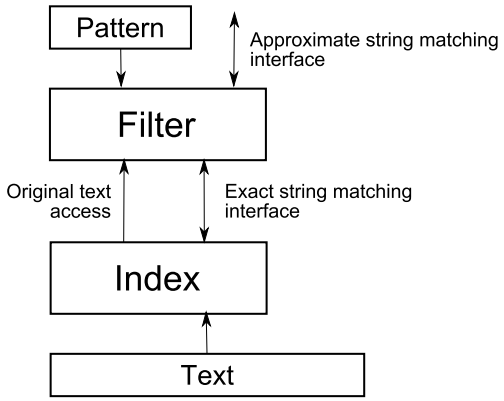
Obrázek 3.17: Situace na grafech ukazuje že, pravděpodobně díky e-value, je pro relativně krátké vzorky (a tím větší α) lepší situace při větší abecedě. Při dostatečně nízké α se projeví vliv velikosti k-okolí, jenž je u filtrace s vyhledáváním jedné chyby stále v rozumné míře. (Pozn.: u $|P| = 20$ je $\alpha = 50\%$ a na $|P| = 100$ je $\alpha = 10\%$)

Kapitola 4

Realizace

4.1 Základní struktura

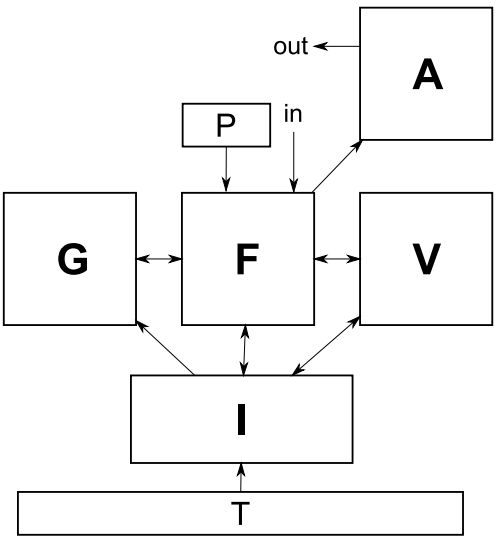
Navržený filtrační algoritmus využívá k vyhledávání předem daný index. Program je však realizován tak, aby bylo možné změnit indexovací algoritmus za jiný. Dokonce pro funkci filtru není důležité, jaký algoritmus pro přesné vyhledávání bude použit a k tomuto účelu může sloužit i některý z on-line algoritmů. Extrakce textu tak může být řešena přímým přístupem k datům textu. Text i vzorek je dán souborem (jeho obsah nemusí být tisknutelný). Tato základní struktura je vyobrazena na obrázku 4.1.



Obrázek 4.1: Znázornění komunikace základních součástí při přibližném vyhledávání.

Základními parametry pro přibližné vyhledávání zadanými uživatelem jsou vzorek, maximální tolerovaný počet chyb (číslo k) a text. Soubor textu na obrázku může představovat již komprimovaný a indexovaný text. Výstupem pro uživatele jsou pak všechny přibližné výskyty vzorku v textu. Vnitřní rozhraní mezi indexem a filtrem je rozebíráno v sekci 4.6.

Aby bylo možné realizovat všechny hlavní přístupy k filtraci (viz sekce 2.3), implementoval jsem zobecněný filtrační algoritmus, jehož nastavením lze mezi těmito přístupy volit. Rozhodl jsem se pro modulární architekturu filtru, jenž je vyobrazena na obrázku 4.2. Implementace programu je tedy rozdělena na moduly, které lze dle potřeby vyměnit, nebo v případě specializovaného použití zcela vypustit.



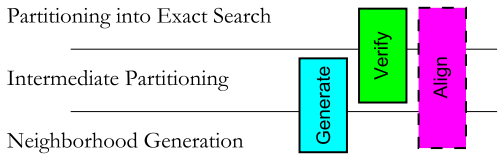
Obrázek 4.2: Propojení jednotlivých modulů: Filtr, Generátor, Verifikátor, Aligner („zarovnávač“) a Index. Moduly jsou blíže popsány v dalších částech této kapitoly.

Při přechodu z obr. 4.1 na obrázek 4.2 je vidět že filtrační algoritmus se skládá z jádra (dále bude nazýváno *Filtr*), jenž zprostředkovává komunikaci všech modulů. Např. zpracovává výsledky přesného vyhledávání, které je prováděno modulem *Index*. Modul *Generátoru* pak provádí generování okolí libovolného řetězce. Pro správnou funkci potřebuje informace o používané abecedě – tato informace může být poskytnuta Indexem. *Verifikátor* provádí ověřování podobnosti určitých částí textu s částmi vzorku. Proto tento modul vyžaduje extrakci libovolné části textu.

Zvláštností mého návrhu je pak další modul pro *Zarovnání*, jenž provádí globální zarovnání („global Alignment“) vzorku s nalezeným výskytem. Slouží k lokalizaci konkrétních chyb v každém výskytu a provádí jeho transformaci do podoby vhodné pro výstup programu uživateli. Výhodou tohoto postupu je že tento modul může použít jiná, složitější, kritéria než jaká byla použita v procesu vyhledávání (podrobně viz sekce 4.4). Smyslem těchto zvláštních kritérií je produkovat vhodného zástupce celé množiny možných zarovnání, resp. indukovaných výskytů.

Při využití algoritmu pro přibližné vyhledávání nemusí být tyto informace

podstatné. Pokud jsou dostačujícími informacemi počátek výskytu, jeho délka a počet chyb (resp. vzdálenost od vzorku), pak může být modul pro zarovnání vynechán. Tato skutečnost je znázorněna na obrázku 4.3 přerušovanou čarou. Stejný obrázek také říká, že pro filtraci s vyhledáváním bez chyb není nutný Generátor, zatímco přístup spočívající na generování k-okolí vzorku nepotřebuje Verifikaci.



Obrázek 4.3: Souvislost využití modulů v závislosti na nastavení programu. Znázorněné moduly a modul Filtru pak nejsou nezbytné pro dotazy kde $k = 0$.

Moduly mohou být měněny kvůli různým implementacím daného algoritmu (není pak třeba měnit kód v dalších modulech) – současná verze implementace obsahuje několik verzí určitých algoritmů, lišící se časovou či paměťovou složitostí. Kód je psán tak, aby bylo možné jednoduše přepnout na jinou implementaci, aniž by byla nutná jakákoliv režie při běhu programu (je ale nutné překompilovat projekt).

Dalším důvodem k rozšíření modulů může být přidávání dalších metrik vzdálenosti – implementovány byly všechny potřebné funkce pro Hammingovu a Levenshteinovu vzdálenost.

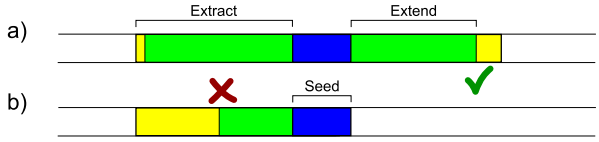
Všechny moduly (včetně Indexu, jehož nejsem autorem) byly implementovány v jazyce *C*. Implementace obsahuje volitelné (formou podmíněného překladu) výpisy měření času určitých úkonů programu. Podporovány jsou systémové funkce OS *Windows* i *Linux*. Testování přeběhlo na obou platformách a také pro 32b i 64b architekturu (primárně na Windows IA-32, sekundárně na Linux x86-64).

4.2 Filtrace

Použil jsem zobecnění všech základních přístupů k filtrování a vzorek je dle nastavení programu dělen na $k + s$ podvzorků (viz sekce 3.5). Dělení je disjunktní a délky podvzorků se liší nejvýše o 1 znak. Pro generování k-okolí podvzorku je vybráno nejmenší k zaručující bezztrátovost (viz sekce 3.1). Ve zvláštních případech samozřejmě je obecný algoritmus méně efektivní než algoritmus který nastavení představuje.

Samotná filtrace probíhá technikou *Seed&Extend*. To znamená, že každý výskyt nalezený Indexem je samostatně zpracován. Tento výskyt se nazývá *seed*, protože se jedná pouze o část vzorku, označení „výskyt“ bude dále použito ve smyslu výskytu celého vzorku. Filtr nejdříve kontroluje, zda je

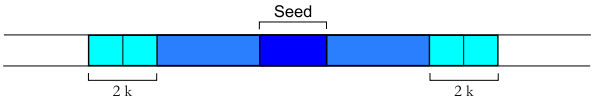
vzhledem k umístění potenciálního výskytu možné, aby šlo o validní či zatím nereportovaný výskyt. Následuje fáze *extend* kdy je provedena verifikace neznámých oblastí výskytu. Nejprve směrem od seedu na jednu stranu a pokud výskyt není zavržen, pak i z druhé strany seedu opačným směrem. Díky implementaci Verifikátoru a těmto opatřením se minimalizuje délka ověřovaného textu. Situace je ilustrována na obrázku 4.4.



Obrázek 4.4: Dva scénáře průběhu ověřování. Pruh symbolizuje text, z něhož jsou části extrahovány (žlutě) a následně verifikovány (zeleně). V případě a) jde o validní výskyt, případ b) je nevalidní. Oproti extrahování a ověřování celé oblasti potenciálního výskytu jsou patrné významné úspory při vykonávání obou operacích.

Aby bylo možné efektivně využít toto částečné ověřování oblasti výskytu, je třeba sledovat počet chyb, aby omezení pro verifikaci bylo co nejpřesnější. Je tedy znám počet chyb v seedu a výsledkem operace extend je v případě pozitivního rozhodnutí také počet chyb v ověřené části výskytu. Odečtením zaznamenaných chyb od k je získán maximální tolerovaný počet chyb v další verifikaci.

Nevýhodou tohoto algoritmu je to, že kontrolován je každý seed i v přídě že je indexem nalezeno více částí téhož výskytu. Aby nebyl stejný výskyt reportován vícekrát, je udržován seznam již reportovaných výskytů. V případě Hammingovy vzdálenosti tak lze ukončit zpracovávání seedu z reportovaného výskytu velmi záhy, již před první extrakcí. Pro Levenshteinovu vzdálenost však nelze předem určit, kde výskyt začíná nebo končí (viz obrázek 4.5). Proto je nejprve spouštěn extend doleva, který zjistí i pozici počátku výskytu. Kvůli bezztrátovosti nelze předpokládat, že výskyty se v textu nebudou překrývat.



Obrázek 4.5: Vymezení oblasti textu k ověření. Modře je vyobrazena oblast potenciálního výskytu v editační vzdálenosti. Světlou barvou je pak vyznačeno rozmezí, kde se může nalézat jeho začátek respektive konec (za předpokladu že seed byl nalezen bez chyby). Poměr ve vyobrazení odpovídá $\alpha = 10\%$.

Nerealizovanou technikou, kterou zde navrhnu je tzv. *Scaffold*, čili lešení. Lešení se používá k udržování podsekvencí mezi nimiž je známo pořadí a relativní pozice, přestože není znám celý obsah lešení. Zpracováním všech výstupů

z Indexu tímto způsobem je pak možné ještě více snížit délku verifikací (podrobněji viz obrázek 4.6). Také odpadá řešení opakované kontroly stejného výskytu – ať validního či nikoli. Dále, na rozdíl od realizovaného algoritmu, lešení umožňuje využít výhod nastavení $s > 1$. Tato technika si však žádá vyšší nároky na režii (především na paměť, neboť ukládá všechny výsledky přesného vyhledávání Indexem). Pro složitější metriky, včetně editační vzdálenosti, je kvůli efektu zmíněnému na předchozím obrázku komplikovanější zařazení nálezu do lešení. Na druhé straně Seed&Extend je velmi jednoduchý a má minimální paměťové požadavky (umožňuje okamžitou spolupráci modulů, neprobíhá po fázích).

Obrázek 4.6: Grafické znázornění lešení. Každé lešení odpovídá potenciálnímu výskytu vzorku. Na příkladu jsou modře vyobrazeny nalezené podvzorky – uložen je zde také počet chyb, s nimiž byly nalezeny. Verifikovány tedy budou jen šedé části, jenž jsou krátké a omezení je vypočítáno z celého lešení. Pro tento výskyt budou díky lešení ověřeny nejvýše jen jednou – bez nutnosti vyhledávání ve seznamu (ne)validních výskytů. Seed&Extend by ve vyobrazeném případě mohl provést (částečnou) verifikaci až třikrát, navíc včetně modrých oblastí.

4.3 Verifikace

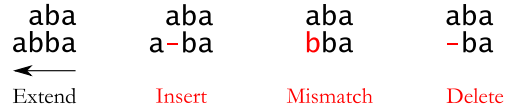
Při popisu filtrace byly v zásadě definovány požadavky na verifikační algoritmus. Vstupem je určité k a dvě sekvence, výstupem pak rozhodnutí zda jsou sekvence vzdáleny o více než k či nikoli. V případě že je počet chyb menší či roven k, pak je požadovaným výstupem i skutečná vzdálenost sekvencí.

Implementace verifikátoru je tedy založena na algoritmu měření vzdálenosti, jenž je přerušen v nejbližším okamžiku kdy je zřejmé že vzdálenost je jistě větší než k. Nízký parametr k tedy má významný positivní vliv na délku běhu algoritmu, která je datově citlivá. Nejhorší případ odpovídá klasickému měření vzdálenosti.

Pro metriky vzdálenosti kde chyba může ovlivnit délku řetězce (tedy např. editační vzdálenost), je však problém o trochu složitější. Extend, na rozdíl od klasické verifikace, má stanovenou počáteční pozici ale není známa pozice, kde ověřovaná oblast končí. Obrázek 4.5 ukazuje, že je možné jen odhadnout rozmezí konců oblasti. To je podstatné pro extrakci, jenž zajistí dekompresi nejmenší části textu avšak dostatečně dlouhé i pro nejdelší případ výskytu.

Výstupem „Extend verifikátoru“ tedy musí být i skutečná hranice výskytu (samozřejmě jen v případě že se sekvence dostatečně shodují). Stanovení této hranice však není jednoznačné – jednoduchý příklad je na obrázku 4.7. Dle

požadavků na program (sekce 1.2) je však vhodné považovat tyto různé výsledky za varianty téhož výskytu. Protože počet různých kombinací může vest k velkému počtu variant, je vhodné reportovat pouze jednu jako zástupce této množiny. Za tímto účelem jsem stanovil heuristiku, která určuje, že velikost oblasti výskytu vypočítaná extend verifikátorem, je vždy ta největší splňující podmínky verifikace. Důvodem tohoto doporučení je to, že později se z výpisu výskytů dají snadno v případě potřeby odvodit i ostatní indukované výskyty. Pokud by byla reportována kratší varianta, nebyly by přímo známy následující znaky textu potřebné k podobnému odvození.



Obrázek 4.7: Extend verifikace s jednou editační chybou extrahované sekvence „abba“. Možné jsou tři interpretace s 1 chybou, pro každou z nich je počátek výskytu na jiné pozici v extrahované sekvenci.

Jak možná vyplývá z předchozího, je nutné rozlišovat mezi *extend doleva* a *extend doprava*, podle toho který okraj výskytu je ověřován. Okraje je, pro zajištění jednoznačnosti, nutné ověřit i v případě, že je dostatečná shoda této oblasti zajištěná seedem (pro Hammingovu vzdálenost v tomto případě stačí k ověření výskytu jedna verifikace). Alternativu by bylo zajištění, že budou kontrolovány nejdříve přesné výskyty nejdelších řetězců z daného okolí zkoumaného podvzorku. Pro Scaffold techniku je možné toto zajistit, jinak zde platí stejné využití Extend verifikátorů (oblasti mezi seedy v lešení jsou pak ověřovány standardním verifikátorem).

Verifikátor pro Hammingovu vzdálenost je implementován jednoduše lineárním průchodem, tedy v čase $\mathcal{O}(m)$ se zanedbatelnými nároky na paměť. Verifikátory pro Levenshteinovu vzdálenost vychází z algoritmu DP, který má časovou i paměťovou složitost $\mathcal{O}(m^2)$. Pro extend je to přesněji $\mathcal{O}(m \cdot (m+k))$, avšak k lze vzhledem k m zanedbat. Při realizaci lze „extend doleva“ převést na problém „extend doprava“ se zrcadlově převrácenými vstupními řetězci, jenž lze řešit DP (funkce je implementována shodně jen rozdílnou indexací znaků řetězců).

Paměťovou složitost jsem vylepšil výpočtem po řádcích. Dále jsem implementoval diagonální[22] verzi algoritmu, jenž využívá znalosti parametrem zadaného k. Díky tomu jsem vylepšil časovou složitost v nejhorším případě na $\mathcal{O}(m \cdot k)$ a paměťovou pouze na $\mathcal{O}(k)$.

[22] Vyčíslován je jen diagonální pás matice.

4.4 Zarovnání

Motivace pro zarovnávací modul byla již uvedena v sekci 4.1. Modul presentuje uživateli výsledky přibližného vyhledávání, konkrétně vypisuje nalezený validní výskyt tak aby se co nejvíce podobal zadanému vzorku. Přínosem je tedy lokalizace jednotlivých chyb a také určení druhu chyby. Tyto informace totiž během filtrování nejsou udržovány.

Další výhodou je možnost nastavení zarovnávání tak aby výsledky co nejlépe odpovídaly dané aplikaci. Využívá se toho, že počet validních výskytů na výstupu programu je značně menší než celkový počet potenciálních výskytů, jenž jsou algoritmem filtrovány. Lze tedy použít jednoduchý a rychlý Filtr pro přibližné vyhledávání a na výsledky aplikovat složitější algoritmus.

Úkolem modulu je také rozlišovat nejednoznačné interpretace rozmístění chyb (zmíněno na obrázku 4.7). Lze tak vybrat nejpravděpodobnější scénář vzniku chyb, v závislosti na aplikaci. Pro editační vzdálenost byla implementována heuristika upřednostňující z chyb nejprve *mismatch*, poté *insert*, a nakonec *delete* (znaku v textu).

Vrátím-li se k motivační úloze z oblasti biologie, zde může být rozložení chyb důležitou informací. Na základě pozorování bylo zjištěno, že osamocený výskyt chyby je pravděpodobně mutace báze. Naopak v sekvenci chyb je pravděpodobnější, že jde o součást podsekvence, která byla do DNA vložena, či z ní odebrána. Při srovnávání dvou sekvencí nemá smysl odlišovat prioritu vkládání oproti odebírání bází, proto se často chyby obou druhů označují *gap* (čili mezera, viz sekce 2.2.4). Pro shluk chyb je však pravděpodobné, že jsou všechny stejného druhu.

Nejrozšířenějším modelem zachycujícím tyto preference je tzv. *afinní skórování*. První chyba typu *gap* (ve smyslu editační operace bývá označena jako *open gap*) je ohodnocena horším skóre než jakýkoliv *mismatch*. Avšak jakákoliv další chyba typu *gap*, jenž bezprostředně následuje po chybě stejného druhu (operace je označována jako *extend gap*), je hodnocena lepším skóre než *mismatch*. V modulu bylo použito skórování operací pro afinní zarovnání shodné s implicitním nastavením algoritmu *blastn-short* z nástrojů BLAST+ [18]:

- *Match* – odměna +1
- *Mismatch* – penalizace −3
- *Open gap* – penalizace −5
- *Extend gap* – penalizace −2

Shodné skórování používá také algoritmus *blastn*, s výjimkou odměny, která byla v BLAST+ změněna na +2 (*blastn* z Legacy BTAST používal také +1). Vypočtené skóre zarovnání by tak mělo být zcela srovnatelné s BLASTem (uvedené hodnoty však lze v kódu jednoduše změnit).

S touto definicí chyb nelze vypočítat optimální výsledek běžným způsobem, pomocí matice DP. Ta totiž představuje hladový algoritmus. Zatímco posouzení chyby jako *gap* je dočasné zhoršení, jenž je správným rozhodnutím pouze v případě, že následuje dostatečný počet chyb stejného druhu. Zda se zhoršení vyplatí, však nelze dopředu znát. Problém je možné vyřešit souběžným vyplňováním třech různých matic DP, každou pro jeden druh chyby. Vzhledem k definici vzdálenosti vlastně lze použít jen dvě matice – proto pro afinní zarovnání byla použita verze jen se 2 maticemi (viz [32]).

Zarovnání pro Hammingovu vzdálenost je triviální – časová složitost je $\mathcal{O}(m)$ a paměťové nároky prakticky žádné. Pro Levenshteinovu vzdálenost byl použit algoritmus Needleman-Wunsch, specializovaný na neváženou editační vzdálenost. Implementována byla jeho diagonální verze, podobně jako v případě Verifikátoru, jenž má časovou složitost jen $\mathcal{O}(m \cdot k)$. Z tohoto důvodu je pro zarovnání také vstupem k, tedy počet chyb ve výskytu vůči vzorku (obdobně jako v případě Verifikátoru to tedy není nutně stejné k jako při zadání dotazu na přibližné vyhledávání). Vzhledem k tomu že afinní skórování nedefinuje optimální zarovnání na základě počtu chyb, nebyla znalost k pro urychlení algoritmu použita. Časová složitost je tedy $\mathcal{O}(m^2)$.

Protože výstupem není jen optimální skóre, ale také příslušné globální zarovnání, je kvůli *backtrackingu* třeba si zapamatovat vypočtenou matici (resp. matice). Zde je vhodné zmínit dva přístupy. První spočívá v uložení matice hodnot skóre, jenž pak musí být při *backtrackingu* znovu porovnáváno. Druhý přístup zaznamenává přímo jednotlivé druhy chyb – při *backtrackingu* pak není třeba provádět žádné výpočty či porovnání. Tento přístup byl implementován pro Levenshteinovu vzdálenost. Informace o chybách (mohou existovat až 3 interpretace chyby) byly uloženy přímo do buněk matice formou bitového pole. Není proto třeba více paměti (maximální počet chyb ale o 3b omezenější). Protože při afinním zarovnání nelze druh chyby určit při vyplňování matice, byl použit naopak první přístup. Implementována byla také kombinace, protože výsledky některých výpočtů je možné do příznaku uložit a není nutné je provádět znovu. Pro všechny přístupy však byl *backtracking* implementován iterativně, s časovou složitostí $\mathcal{O}(m + k)$.

Aby při každém zarovnání nebylo nutné alokovat a dealokovat potřebné matice, je modul vybaven maticemi, jenž jsou alokovány na začátku vyhledávání a dealokovány až po zarovnání všech výskytů. Tyto matice mohou být sdíleny různými zarovnávacími algoritmy. Využito je toho, že výskyty mají velice podobné délky (vzorek svou délku nemění). Paměťová složitost tedy může být předem omezena na $\mathcal{O}(m \cdot (m + k))$. Podobný systém alokace může být použit i v jiných modulech.

Protože modul vypisuje data textu a vzorku, je vhodné zmínit, že tato data jsou před výpisem převedena na zobrazitelné symboly. Řídící znaky se tedy zobrazují jako písmena – bez pomocných *escape symbolů*, protože použití více znaků k reprezentaci jednoho by narušilo vizuální prezentaci zarovnání.

4.5 Generování okolí

Generátor okolí řetězce zprostředkovává jeho vyhledávání s chybami (viz sekce 2.3). Typicky je využit k redukci problému přibližného vyhledávání. Redukcí je snížena délka vzorku (na podvzorky) a podobně je snížena i hodnota k. Proto bude bez újmy na obecnosti popisováno generování k-okolí podvzorku.

Obecný je i proces vyhledávání s chybami tím, že je generováno okolí podvzorku a tyto jednotlivé řetězce vyhledávány Indexem nezávisle na jeho implementaci. Specializované metody, např. pro index založený na *suffix trie* mohou spojit generování a vyhledávání do jedné operace. Suffix trie totiž umožňuje vyhledat všechny společné prefixy řetězců okolí najednou (viz [13]).

Pro implementaci generátoru je vhodné, aby zkonstruoval všechny řetězce z k-okolí, přičemž nejlépe žádný dvakrát a také omezit potřebu kontroly nevalidních řetězců. Teoreticky bývá generování často definováno jako generování všech řetězců vhodné délky, které jsou následně filtrovány dle vzdálenosti od podvzorku.

Proto jsem při implementaci generátoru pro Hammingovu vzdálenost využil toho, že množinu okolí lze kombinatoricky definovat. Tím je zajištěno generování právě k-okolí. Při konstrukci řetězců jsem dále využil jejich podobnosti, generovány a modifikovány jsou tedy jen znaky obsahující chyby. Paměťová složitost je proto jen $\mathcal{O}(k)$. Časovou složitost lze shora omezit funkcí $\mathcal{O}(k \cdot m^k \cdot \sigma^k)$, nicméně skutečná složitost je nižší protože velikosti podokolí nejsou stejné (viz graf na obrázku 3.9) a algoritmus má složitost odpovídající velikosti k-okolí.

Předchozí věta naráží na skutečnost, že algoritmus generuje potřebná k-podokolí. V původním návrhu filtrace jsem se také zabýval možností využití vyhledávání podokolí s vysokým k, namísto vyhledávání některých podokolí s nižším k jejichž mohutnost je vyšší. Ačkoli tyto myšlenky nebyly realizovány, algoritmus je možno použít ke generování jen určitých podokolí.

Implementoval jsem také generátor pro 1-okolí, jenž má narozdíl od obecného rekurzivního algoritmu výrazně nižší režii. Důvodem byl předpoklad, že nejčastějším požadavkem bude generování okolí s velmi nízkým k (viz sekce 3.4). Součástí implementace je proto jednoduchý algoritmus, který na základě požadavku deleguje generování na funkci s nejnižší režií.

Pro realizaci generátoru pro Levenshteinovu vzdálenost již kombinatorický přístup není tak vhodný. Použil jsem poměrně obecný přístup založený na výpočtu vzdálenosti všech řetězců (relevantních délek) k podvzorku pomocí DP. To zaručuje, že žádný řetězec nebude vygenerován více než jednou a také že žádný nemůže být opomenut. Opět jsem využil podobnosti řetězců a matice DP je vypočítávána pro každý prefix pouze jednou. Navíc jsem použil diagonální algoritmus DP a generátor díky vyhodnocení prefixu zbytečně negeneruje (a neměří) řetězce jenž mají vzdálenost jistě větší než k. Paměťová náročnost je tak $\mathcal{O}(m \cdot k)$, lze-li zanedbat k vzhledem k m. Časová složitost je $\mathcal{O}(k \cdot \sigma^{m+k})$, kde k zanedbáno není, ale není zohledněn vliv ořezávání.

4.5.1 Využití trie

Modul je realizován tak, že každý vygenerovaný řetězec může být rovnou vyhledán Indexem a výsledky zpracovány Filtrem. Proto je používána výše popsaná filtrace „Seed&Extend“. Tento návrh tedy umožňuje proudové zpracovávání. Implementovaná aplikace je jednovláknová, možným vylepšením je proto jednotlivé moduly realizovat jako vlákna a využít takto výhody tohoto přístupu. Další výhodou je minimální spotřeba paměti.

Modul však podporuje i druhý přístup, založený na uložení generovaného k-okolí. Velmi snadno lze modul přepnout do módu, kdy jsou generované řetězce ukládány do trie. Zřejmou nevýhodou je spotřeba paměti, jenž tuto metodu limituje jen pro vhodné nastavení. Výhodou je pak možnost snížení počtu vyhledávacích dotazů pro Index.

Nejsou totiž díky trie vyhledávány dvakrát stejné řetězce. Tato situace může nastat pro velmi repetitivní vzorky. Pokud jsou si jednotlivé podvzorky podobné, je pravděpodobné že jejich okolí se bude překrývat. V případě kdy by např. dva podvzorky byly totožné, lze ušetřit vyhledávání celého okolí druhého podvzorku.

Takové využití samozřejmě předpokládá, že do trie budou generovány okolí všech podvzorků. Nelze ovšem sjednocením shodných řetězců ztratit informaci o tom, z okolí kterého podvzorku řetězec může pocházet. Dále jsem zatím nezdůraznil, že výstupem generátoru je pro každý řetězec z okolí také jeho vzdálenost k danému podvzorku. Tyto informace jsou důležité pro bezztrátovost a výkonnost filtrace. Trie proto v koncových uzlech obsahuje seznam těchto čísel, jenž je předán Filtru. Filtr pak může zvážit nalezené výskyty např. jako výskyty podvzorku p_1 s k_1 chybami, i jako výskyty podvzorku p_2 s k_2 chybami.

Další výhodou uložení okolí do trie, je možnost jeho ořezávání. Není tak například nutné vyhledávat řetězce, pro které je z výsledků předchozích vyhledávání možné odvodit, že se v textu nenalézají. Proto je při procházení trie vyžadována zpětná vazba od modulu, jenž provádí vyhledávání. Implementace umožňuje jednoduše v kódu strategie ořezávání vyměnit.

Implicitně je používána strategie, která vyhledává nejdříve vždy nejkratší prefix, jenž je platným řetězcem z okolí. Pokud není nalezen žádný výskyt, není dále podstrom zpracováván, neboť se jistě žádný řetězec z tohoto podstromu nemůže v textu vyskytovat. Alternativou je naopak vyhledávání platných řetězců nejprve z listů trie. Pokud řetězec není nalezen, není nutné vyhledávat jeho prefixy.

Je také možné vyhledávat pouze platné řetězce z listů [25]. Taková množina řetězců se nazývá *kondenzované k-okolí*. Z výskytů těchto řetězců pak lze snadno odvodit výskyty jejich prefixů. Problémem je, že ne všechny výskyty. Nejsou totiž typicky vyhledávány všechny řetězce dané délky, jenž obsahují tento prefix. Listů je samozřejmě více než nejkratších platných prefixů, ale technika těží spíše z nižších e-value delších řetězců.

4.5.2 Využití abecedy

Aby mohl generátor fungovat, je třeba znát abecedu, z níž budou chyby generovány. Abeceda indexovaného textu je předána modulu generátoru při jeho startu. Používána je také při konstrukci a procházení trie, neboť tato struktura je implementována tak, že neobsahuje vlastní znaky řetězců.

Protože velikost okolí závisí na velikosti abecedy (značeno σ), realizoval jsem volitelné vylepšení, jenž vyhledávání urychluje. Vylepšení je založeno na zúžení abecedy, proto z principu nezaručuje bezztrátovost. Existuje míra *IPM*, jenž udává převrácenou hodnotu pravděpodobnosti, že dva náhodně vybrané znaky z textu se shodují. Jde tedy o veličinu podobnou entropii a udává kolik znaků z abecedy je aktivně využíváno. Například pro samostatné sekvence DNA je typicky IPM = 4.000 zatímco $\sigma = 15$ [33]. Jedenáct znaků abecedy se tedy vyskytuje poměrně vzácně.

Vylepšení spočívá v použití abecedy vzorku. Lze předpokládat, že ve vzorku se objeví všechny znaky, jenž jsou velmi časté a naopak je malá pravděpodobnost že bude obsahovat znaky vzácné. Vyhledávání řetězců se vzácnými znaky s velkou pravděpodobností nezaznamená žádný výskyt, proto ztrátovost tohoto postupu je dostatečně malá. Předpokladem tedy je, že σ vzorku bude odpovídat IPM textu.

Při úloze vyhledávání v DNA tento postup považuji za velmi užitečný (předpoklad je běžně splněn). Bezztrátovost je možné zaručit nastavením algoritmu $s \geq 1$. Avšak i pro menší hodnoty s jsou ztráty minimální, protože nenalezeny jsou jen výskyty, kde se chyba v podobě vzácného znaku vyskytne v oblasti každého podvzorku minimálně jednou, což je pro vzácné znaky nepravděpodobné.

Dále byl řešen problém, kdy je vyhledáván vzorek obsahující znak, jenž se nevyskytuje v textu. Může jít o znak natolik vzácný (jeho existence nemohla být dedukována Indexem), nebo může jít o vadu vstupu (tento problém lze očekávat při vyhledávání v přirozeném jazyce, např. vyhledávání s diakritikou v textu, který ji neobsahuje). Rozdílná abeceda by způsobila chybu programu, proto Filtr využívá také znalosti abecedy textu a nejdříve ze vzorku odstraní neznámé symboly.

Odstranění je realizováno nahrazením speciálním znakem s významem „neznámý symbol“. Index může být schopen tento speciální znak zpracovávat (při realizaci bylo využito toho, že indexovací algoritmus interně takový symbol používá). Pokud je využit trie, pak Generátor řetězce obsahující tento symbol neukládá. Nejsou tedy ani vyhledávány Indexem. Připomeňme, že řetězce obsahující neznámý symbol nemohou být v textu nalezeny, tímto postupem proto nedochází k žádným ztrátám.

Kromě řešení problému rozdílných abeced, je možné tuto vlastnost cíleně využít. Při vyhledávání v DNA mohou být využívány tzv. nejednoznačné báze. Tyto znaky jsou posuzovány jako neshodné s jakýmkoliv znakem, včetně sebe sama [19]. Tuto vlastnost nelze řešit jen na úrovni verifikace a zarovnání,

protože při procesu filtrace a generování je počet chyb v řetězci důležitým parametrem, jenž by nebyl správně stanoven. Díky zavedení neznámého symbolu je možné ošetření této výjimky převést na Verifikátor (a modul pro zarovnání). Je vhodné poznamenat, že tento speciální znak se v textu vyskytovat může, pokud je znám jeho význam. V praxi se v sekvencích DNA používá k tomuto účelu písmeno '`N`' (realizovaný algoritmus je však obecný a proto tomuto písmenu, pokud se v textu vyskytne, není přisuzován zvláštní význam).

4.6 Indexace

Modul indexu zprostředkovává práci s původním textem. Indexovací algoritmus je možné nahradit jiným – modulem se může stát také on-line vyhledávací algoritmus. Je jen třeba splnit následující rozhraní.

Otevření souboru. Operace slouží k inicializaci algoritmu a načtení (indexového) souboru textu. Vstupem je název originálního souboru, zbytek programu tedy není vázán na typ algoritmu. V případě zpracovávání komprimovaných či indexovaných textů se předpokládá, že modul připojí ke jménu souboru určitou příponu. Je tedy možné používat více různých indexových souborů. Není podmínkou, aby původní soubor textu na daném umístění existoval, pokud zde existují soubory nutné pro funkci modulu. Výstupem této funkce je délka původního textu.

Během zpracování dotazu na přibližné vyhledávání je následně vyvolána řada dotazů na modul indexu. Soubor je proto načten jen jednou, a všechny následné operace jsou prováděny na načteném textu. Nad jedním textem může být provedeno více přibližných vyhledávání. Po dokončení přibližného vyhledávání nad daný textem je zavolána funkce *uzavření souboru*, jenž slouží k deinicializaci algoritmu.

Vytvoření indexového souboru. Tato operace je alternativou k *otevření souboru* (má stejný vstup), po jejím dokončení se tedy očekává stejný stav (modul je připraven vyhledávat v textu). Tato funkce je volána pokud je text zpracováván poprvé. Off-line algoritmy tedy musí vytvořit komprimované či indexové soubory, jenž budou následně využívány.

Zásadní operací je *přesné vyhledávání.* Vstupem je řetězec, jenž má být vyhledán a výstupem je pak seznam všech výskytů tohoto řetězce v původním textu. Seznam by měl být reprezentován polem čísel. Každé číslo označuje pozici právě jednoho výskytu v původním textu. Tato čísla nemusí být seřazená. Součástí výstupu je dále i počet nalezených výskytů.

Druhou často využívanou operací je *extrakce.* Vstupem je pozice v textu a délka podřetězce. Výstupem by měl být takto určený podřetězec původního textu. Filtr zajišťuje, že argumenty dotazu mají validní hodnoty. Extrahovány jsou řetězce nejvýše délky vzorku, nejméně délky podvzorku.

Vzhledem k tomu že filtrační algoritmus je obecný, nelze řetězce reprezentovat formou *null-terminated*, neboť znak může být reprezentován jakýmkoliv

číslem. Hodnoty znaku jsou implementací omezeny na rozsah `unsigned char`, nicméně tato reprezentace lze velmi jednoduše změnit na jiný datový typ. Součástí reprezentace řetězce je tedy i číslo udávající jeho délku. Délky a pozice jsou také vyjádřeny snadno vyměnitelným datovým typem. Je-li to možné, doporučuji použít `unsigned long`, aby bylo možné zpracovávat i opravdu velké texty a Filtr je implementován s předpokladem, že použitý datový typ není schopen reprezentovat záporná čísla. Filtrační algoritmus se tedy může přizpůsobit modulu indexu (řešení aktuální realizace je popsáno v sekci 4.6.2).

Pokud není program upraven tak, aby nebylo třeba modulu Generátoru, pak je nutné implementovat i funkci *získání abecedy*. Důvody jsou popsány v sekci 4.5.2. Výstupem funkce je abeceda používaná původním textem. Reprezentována je jako řetězec všech použitých symbolů. Výstupem je také tzv. speciální symbol, jenž se v textu nevyskytuje. Jeho existence poskytuje výhody popsané ve výše zmíněné sekci, jenž jsou současnou verzí podporovány, avšak program je možno s minimálním úsilím upravit tak, že tento symbol nebude mít žádný zvláštní význam. Konkrétní hodnoty znaků a jejich pořadí nejsou pro funkci důležité, ale pro filtrační algoritmus typu „Neighborhood Generation“ jsou výskyty reportovány v abecedním pořadí – jenž je vyvozeno z této reprezentace abecedy.

V současné verzi jsou součástí modulu také funkce nezávislé na algoritmu pro práci s textem. Jedná se o načtení souboru vzorku, či funkce pro bezpečný výpis řetězců (neboť symboly mohou být i řídící znaky).

4.6.1 FM-Index

Jako indexovací algoritmus byl v mém programu použit *FM-index verze 2* (zkráceně *FMIv2*), jehož autory jsou Paolo Ferragina a Rossano Venturini. Tato implementace splňuje aplikační rozhraní pro vlastní indexy doporučené portálem *Pizza&Chili*[23], jenž se právě těmito algoritmy zbývá. Zdrojové kódy jsou dostupné na tomto portálu. Toto rozhraní v zásadě splňuje výše uvedené požadavky (jen neobsahuje získání abecedy), nemělo by proto být problematické využít jiný algoritmus, jehož implementace toto doporučení respektuje.

Je třeba doplnit, že FMIv2 využívá ke své funkci další algoritmus. Jeho plný název je *Deep Shallow Suffix Sort* (častěji uváděn jako *ds_ssort*) a autorem je Giovanni Manzini. Tento algoritmus slouží k rychlé konstrukci *Suffix Array*.

FMIv2 využívá jistý druh *compressed suffix array*, aby mohl být soubor komprimován a přitom aby složitost vyhledávání nebyla výrazně zhoršena (oproti indexům *suffix tree*, nebo *suffix array*) [3]. Zjištění počtu výskytů má složitost úměrnou velikosti m, nezávisí tedy na n [3]. Přesné vyhledání každého z výskytů je provedeno v čase $\mathcal{O}(\log^c(n))$, kde c je kladná konstanta zvolená při vytváření indexu [3].

[23] `http://pizzachili.dcc.uchile.cl/` nebo `http://pizzachili.di.unipi.it/`

Hlavní rozdíl oproti původnímu *FM-indexu* (tedy verzi 1) je v tzv. značkování textu. Tento proces vkládá do původního textu nové symboly (jedná se o *speciální symbol*, jak byl definován výše, tedy takový který se jinak v textu nevyskytuje). Tyto symboly jsou v textu rozmístěny pravidelně. Při následném provádění *Burrows-Wheelerovy transformace*, která je založena na lexikografickém seřazení všech cyklických rotací textu, jsou všechny speciální symboly seřazeny k sobě. Pro urychlení operace vyhledávání je pro tyto znaky uložena jejich pozice v textu [3].

Frekvence značkování je jedním z nastavitelných parametrů FMIv2. Ve své implementaci jsem použil výchozí nastavení algoritmu. Nastavení je případně možné snadno změnit v kódu modulu, není ale přístupné uživateli.

Nicméně samotný fakt že algoritmus používá speciální symbol pro značkování, poněkud omezuje jeho použití. Pro binární soubory jsou typicky použity všechny přípustné hodnoty k reprezentaci znaku a není proto možné přidat nový symbol. Ačkoli by měl algoritmus fungovat i v tomto případě (značkováním všech pozic), toto použití nedoporučuji.

4.6.2 Úpravy FMIv2

Rozhraní FMIv2 bylo doplněno o chybějící funkci pro získání abecedy. Její složitost je minimální, neboť potřebné informace jsou algoritmem interně využívány a byly proto uchovávány v indexu (včetně speciálního symbolu).

Musel jsem však provézt mnohem více úprav zdrojového kódu, neboť jsem odhalil jisté chyby v původní implementaci (soubory, jenž nebyly nezbytné pro funkci indexu, byly z projektu vyřazeny, jejich obsahu se dále nebudu věnovat). V následujících odstavcích popíši některá zjištění, v případě těch závažnějších také včetně způsobu jakým jsem problémy řešil.

FMIv2 používal pro malé[24] soubory algoritmus *Boyer-Moore*. Důvodem zřejmě bylo, že indexovací algoritmus byl na malých souborech málo efektivní (indexový soubor může být v takovém případě větší než původní text). Kvůli tomu pak na malé soubory nemohla být aplikována extrace, ani nebyla zjišťována abeceda textu. Změnil jsem tedy chování tak, že je FMIv2 použit pro všechny[25] velikosti vstupního textu. V implementaci Boyer-Mooreova algoritmu byla navíc nalezena chyba (operace maxima byla implementována jako minimum).

Stávalo se, že byla extrahována špatná část textu. Díky tomuto posunu mohl *LF-mapping*, což je proces při kterém je řetězec konstruován od svého posledního znaku směrem k prvnímu, podtéct text a způsobit běhovou chybu. Dále jsem opravil kontroly mezí během extrakce – mnoho z chyb bylo možno řešit jiným pořadím výkonných částí kódu. Další příčinou byly výpočty obsahující délku původního textu, zatímco zapracovávaný text obsahoval i značky

[24] Soubor byl považován za „malý“ do 50 kB, „zvláště malý“ do 1 kB.

[25] Správné chování je zaručeno pro texty delší než 2 B.

a byl proto delší. Přidal jsem také možnost extrakce velmi krátkých řetězců (jediný znak).

Závažnou chybou extrakce jsem také objevil v případech, kdy extrahovaný text končí poblíž konce textu (konkrétně v oblasti posledních 64 znaků). V takovém případě byla extrahována zcela jiná část textu. Problém je způsoben lokalizací poslední značky (protože jsou rozmístěny rovnoměrně, zatímco délka textu není nutně dělitelná vzdáleností mezi značkami). Z časových důvodů tato chyba zůstává neopravena, situaci jsem vyřešil jednodušeji. V modulu k indexaci je vstupní text prodloužen (o 64 znaků, tato hodnota je odvozena z výchozího nastavení algoritmu). Prodloužení nezvýší velikost abecedy a není mimo modul zřejmé, způsobí však, že extrakce závěru textu je provedena správně, neboť se za koncem textu značka vždy vyskytuje (s běžným odstupem).

Ačkoli proces vyhledávání nebyl prověřen stejně důkladně jako extrakce (domnívám se, že vyhledávání bylo věnováno při vývoji více pozornosti), chyba byla nalezena i zde. V některých případech mohlo dojít při vyhledávání k modifikaci vyhledávaného vzorku. Následné použití vstupního řetězce pak vede k dalším chybám. Algoritmus totiž používá dočasnou modifikaci vstupních parametrů (mimochodem stejná technika byla použita v implementaci Generátoru, aby nebylo nutné alokovat další paměť a kopírovat řetězce). Postup obnovy vstupního řetězce byl opraven.

Během testování na různých architekturách jsem objevil poměrně vážný nedostatek. Příčinou byla nekompatibilita s 64b architekturou, která způsobila běhovou chybu. Problém je ve vazbě mezi FMIv2 a ds_ssortem. Implementace ds_sortu je vázaná na využití 32b datového typu. Situaci jsem vyřešil tak, že jsem datový typ určený pro vyjádření pozic znaků převedl na 32b `unsigned int`. Některé GNU knihovny znemožňují redefinování datového typu, ale problém byl vyřešen i pro tyto případy.

Funkce algoritmu jsou tedy omezeny 32b adresací i pro platformy schopné zpracovávat řádově větší soubory. V současné implementaci tedy doporučuji prohledávat pouze soubory do velikosti 2 GB[26]. Alternativním řešením může být nahradit ds_ssort jinou implementací, pak je filtr připraven zpracovávat soubory prakticky neomezené velikosti.

[26]Přesněji do 2 GiB (pro vyjádření velikosti souboru ve své práci vždy uvažuji jednotky ve významu násobků 1024 – to odpovídá standardu *JEDEC*).

Kapitola 5

Experimenty

5.1 Popis měření

Vzhledem k motivaci uvedené na začátku práce, se měření zaměřuje na vyhledávání v DNA. Bylo ale provedeno i několik experimentů ke srovnání s vyhledáváním v přirozeném jazyce. Použitá data jsou blíže popsána v příslušných sekcích.

Vyhledávané vzorky různých délek byly vytvořeny jako prefix prohledávaného textu (tedy prvních $|P|$ znaků textu). V experimentech srovnávajících různé abecedy byly vzorky naopak sufixem textu (tedy posledních $|P|$ znaků). Důvodem bylo to, že v přirozeném jazyce bylo na začátku textu mnoho formátovacích znaků, zatímco závěr lépe odpovídal textu běžnému textu. Vzorky byly vytvořeny tímto způsobem, aby bylo zajištěno, že vyhledán musí být nejméně jeden výskyt při libovolném k.

V případě experimentu srovnávající různé délky textu, byly použity prefixy různých délek téhož vstupního textu.

V realizovaném programu jsem implementoval multiplatformní měření času. To umožnilo změřit např. jen čas potřebný k vyhledávání, bez započítání času k načítání indexového souboru a jeho dealokaci po ukončení vyhledávání. Měřen byl strojový čas věnovaný procesu. Ve výsledcích je uváděn tzv. *user time*. Měřen byl i tzv. *kernel time*, jeho hodnoty však byly nízké, či neměřitelné, proto není uváděn.

Při měření času vyhledávání implementovaného algoritmu jsem pro zvýšení přesnosti použil opakované vyhledávání stejného dotazu. Jedno měření typicky měřilo čas 100 opakování vyhledávání, z něhož byl vypočítán průměr. Tento proces byl v rámci jednoho spuštění programu desetkrát opakován a jako výsledek byla použita nejnižší hodnota průměrného času vyhledávání. V případech kdy byl očekáván extrémně dlouhý běh vyhledávání, bylo 100 opakování redukováno na 10 popř. na 1. Tyto situace bylo možné předpovědět pomocí modelu představeném v sekci 3.5.

Pro BLAST však tento postup nebylo možné použít. V experimentech srovnání realizovaného programu s BLASTem byly použity pro měření času informace z programu *GNU time*. Bylo měřeno 100 spuštění programu a z naměřených hodnot vybráno minimum. V jednom případě bylo minimum pod hranicí rozlišitelnosti měřicího programu, v tomto případě byla použita průměrná hodnota. V experimentu sledujícím indexování textu, jenž je časově náročnější, bylo provedeno jen 10 opakování.

V některých experimentech byla sledována spotřeba operační paměti. Pro každé spuštění programu je měřena maximální hodnota spotřeby během běhu programu. Z jednotlivých spuštění pak byl vypočítán průměr. Tato měření byla provedena programem *GNU time*, který sledování paměti podporuje[27]. Měřen byl i počet výpadků stránek, ale hodnoty byly úměrné spotřebované paměti a nebyla zajištěna shodná velikost stránek pro všechny procesy, nejsou proto tyto hodnoty uváděny.

Testovací měření proběhlo na 32b platformě *Windows*. Měření pak proběhlo na 64b architektuře s *GNU/Linux* – z tohoto měření pocházejí uvedené výsledky experimentů. Použitý stroj měl OS *Gentoo 4.8.4*, 16 GB operační paměti a procesor *Intel® Core™ i7-4770* (jádra taktována na 3,4 GHz).

5.2 Měření implementovaného řešení

5.2.1 Použitá data

Ve většině experimentů je použit jako text soubor *Escherichia_Coli*, popsaný níže. Výjimkou jsou jen experimenty v sekci 5.2.3. Tyto experimenty jsou zaměřeny na srovnání textů různého druhu. Konkrétně vyhledávání v sekvenci DNA je porovnáváno s vyhledáváním v textu v přirozeném jazyce. Aby bylo srovnání co nejlepší, byly vybrány soubory *dna.100MB* a *english.100MB*, jenž mají shodnou velikost a byly sestaveny stejným způsobem.

Tyto tři soubory byly získány z *Pizza&Chili korpusu*, jenž je zaměřen na testování algoritmů pro indexování (a kompresi) textu a vyhledávání v takto indexovaném textu [34]. Vybrané soubory jsou složeny z reálných dat.

Escherichia_Coli – Soubor je součástí reálných dat *Repetitivního korpusu*[28]. Obsahuje 23 různých sekvencí genomu bakterie Escherichia Coli. DNA sekvence byly získány z NCBI. Jednotlivé holé sekvence jsou sřetězeny bez jakýchkoliv oddělovačů. Vlastnosti textu [33]:

- Délka textu: 107 MB, $n = 112689515$
- Velikost abecedy: $\sigma = 15$, IPM = 4,000

[27] *BASH time*, kdy je „time“ interpretován jako klíčové slovo, nebyl v experimentech použit. Tato verze, integrovaná v interpretu BASH, měří jen čas.

[28] `http://pizzachili.dcc.uchile.cl/repcorpus/real/`

- Entropie[29]: $H_0 = 2$ b (25,00 %)

dna.100MB – 100MB prefix souboru reprezentativních DNA sekvencí[30]. Použity jsou opět holé sekvence bez anotací, avšak jednotlivé sekvence jsou odděleny koncem řádku. Samotné genetické informace byly získány z *Projektu Gutenberg*. Vlastnosti textu [34]:

- Délka textu: 100 MB, $n = 104857600$
- Velikost abecedy: $\sigma = 16$, IPM = 3,91
- Entropie: $H_0 = 1,977$ b (24,71 %)

english.100MB – 100MB prefix souboru reprezentativních textů v přirozeném jazyce[31]. Použity byly vybrané texty v angličtině z kolekce *Projektu Gutenberg*. Podobně jako v předchozích případech byly odstraněny hlavičky Projektu Gutenberg a soubor je sřetězením jen vlastních textů jednotlivých knih. Vlastnosti textu [34]:

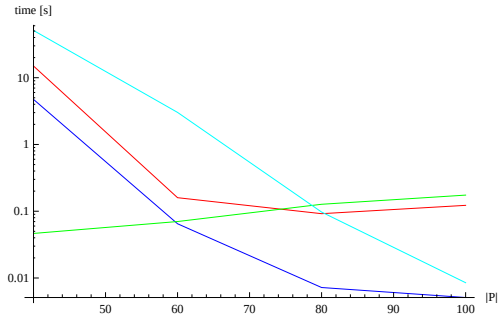
- Délka textu: 100 MB, $n = 104857600$
- Velikost abecedy[32]: $\sigma = 215$, IPM = 15,25
- Entropie: $H_0 = 4,556$ b (56,95 %)

5.2.2 Vliv chybového poměru

Experiment, jehož výsledek je vyobrazen na grafu na obr. 5.1, je zaměřen na to, zda matematický model představený v sekci 3.5 souhlasí s chováním programu. Vyhledávání bylo provedeno vždy s $k = 5$ chybami v Hammingově vzdálenosti. Délka vzorku byla měněna, proto lze pozorovat také vliv různého α. Stejným způsobem byly vytvářeny grafy ve zmíněné kapitole. Délka vzorku byla 40–100 znaků, s krokem 20 znaků. Chybový poměr je tedy mezi $12,5\%$ a 5%. Použity jsou dvě nastavení filtru s vyhledáváním bez chyb, z nichž by měl lepších výsledků dosahovat algoritmus s $s = 1$. Dvě nastavení jsou porovnávána také pro filtraci s přibližným vyhledáváním. Doporučení v sekci 3.5 preferuje nastavení $s = -2$.

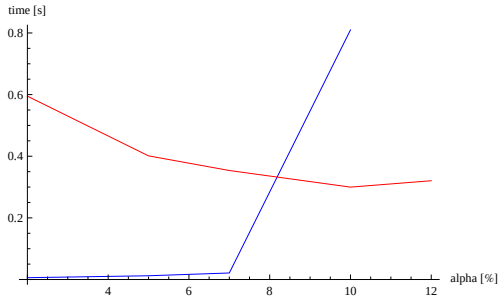
Graf na obrázku 5.1 ukazuje, že chování programu odpovídá modelu. Dokonce jasně potvrzuje předpověď, že pro vysoké α a krátké vzorky muže být přístup vyhledávání podvzorků s chybami lepší než vyhledávání bez chyb. Výsledky také souhlasí s tím, že v opačné situaci je naopak rychlejší vyhledávat podvzorky bez chyb.

[29] Uvedena je empirická entropie 0-tého řádu a v závorce odpovídající kompresní poměr.
[30] `http://pizzachili.dcc.uchile.cl/texts/dna/`
[31] `http://pizzachili.dcc.uchile.cl/texts/nlang/`
[32] Hodnoty σ jsem změřil, převzaté hodnoty IPM doporučuji chápat jako orientační.



Obrázek 5.1: Logaritmický graf času vyhledávání pro různé algoritmy. Modře *Partitioning into Exact Search*, tmavě pro $s = 1$, světle pro $s = 3$. Červeně *Intermediate Partitioning* pro $s = 0$ a zeleně pro $s = -2$.

V dalším experimentu je proto sledováno, jaký je vliv chybového poměru na to, který přístup bude rychlejší. Pro oba přístupy je použito nastavení dle doporučení. Vyhledáván je vzorek délky $|P| = 100$ a měněn je počet tolerovaných chyb, tentokrát však v editační vzdálenosti. Tímto způsobem je docíleno těchto hodnot α: 2%, 5%, 7%, 10% a 12%.



Obrázek 5.2: Graf zobrazuje modře *Partitioning into Exact Search* pro $s = 1$ a červeně *Intermediate Partitioning* nastavený na vyhledávání podvzorků s jednou chybou.

Graf na obr. 5.2 jasně ukazuje kde je pro daný text bod zlomu. Pro $\alpha > 10\%$ se vyhledávání bez chyb stává velmi neefektivní. Jeho čas pro $\alpha = 12\%$ nebyl na platformě Linux naměřen kvůli běhové chybě, jejíž příčina z časových důvodů nebyla zjišťována. Z měření na platformě Windows, kde se žádný problém neobjevil, však vyplývá, že čas tento čas by byl desetkrát vyšší než pro $\alpha = 10\%$, jenž je nejvyšším bodem zobrazeným na grafu.

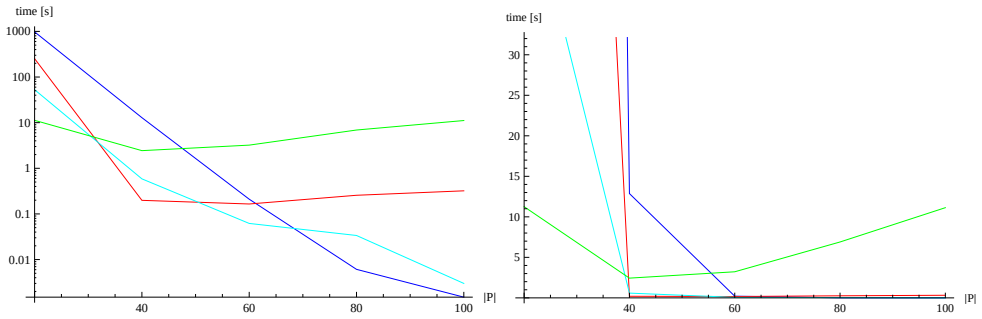
5.2.3 Vliv abacedy

Na grafu na obr. 5.3 jsou výsledky měření vlivu druhu textu na čas vyhledávání. Vyhledávání v DNA je porovnáváno s vyhledáváním v přirozeném jazyce,

konkrétně v anglickém jazyce. Texty se liší především velikostí abecedy. Pro DNA je $\sigma = 16$, zatímco pro angličtinu je $\sigma = 215$.

Vyhledávané vzorky byly vždy ve stejném jazyce jako prohledávaný text. Délky textů i vzorků bylo pro oba jazyky shodné. Vyhledáváno bylo vždy s $k = 5$ editačními chybami, zatímco délka vzorku byla měněna v rozmezí 20–100 znaků s krokem 20. S délkou vzorku tedy klesá chybový poměr, jeho nejvyšší sledovaná hodnota je 25 %.

Testovány byly dva hlavní přístupy (s parametrem dle doporučení). Tedy filtrace s vyhledáváním bez chyb ($s = 1$) a filtrace s vyhledáním s jednou chybou ($s = -2$).



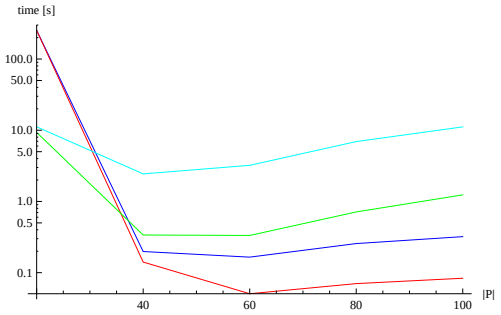
Obrázek 5.3: Logaritmický graf času vyhledávání pro různé algoritmy (vlevo). Vpravo jsou stejná data v lineárním měřítku. Modře je vynesen *Partitioning into Exact Search*, tmavě pro DNA, světle pro angličtinu. Červeně *Intermediate Partitioning* pro DNA a zeleně pro angličtinu.

Graf v logaritmickém měřítku na obrázku 5.3 dobře ukazuje, že vyšší hodnota σ zapříčiňuje pomalejší vyhledávání delších vzorků. Zvláště na grafu v lineárním měřítku je ale patrné, že vyhledávání kratších vzorků bylo pro vysoké σ mnohem rychlejší. Důvodem je především to, že pro $|P| = 20$ je nalezeno více výskytů než v případě delších vzorků. Pro DNA je však tento nárůst více než 1000-krát vyšší než v přirozeném jazyce. Proto je ke grafu vhodné dodat, že pro $|P| = 20$ byl v angličtině každý výskyt nalezen průměrně za 1,023 s. Zatímco v DNA byl každý z výskytů vyhledán v průměru za 22 ms. Uvedené časy platí pro filtraci s vyhledáváním podvzorku s jednou chybou.

Poměrně zajímavým výsledkem je také to jak se mění bod, ve kterém jsou oba hlavní přístupy rovnocenné. Pro DNA se Intermediate Partitioning vyplatil pro nezanedbatelně delší vzorky než při použití v angličtině.

V sekci 4.5.2 jsem navrhl zrychlení vyhledávání založené na zúžení abecedy. Proto jsem část předchozího experimentu zopakoval s tímto vylepšením. Vylepšení má efekt při použití modulu Generátoru, proto jsem sledoval jen algoritmus $s = -2$. Obrázek 5.4 ukazuje naměřené srovnání.

V případě DNA byla abeceda zúžena z 16 na $\sigma = 4$, což odpovídá IPM. Pro anglický jazyk se velikost zúžené abecedy měnila v závislosti na velikosti



Obrázek 5.4: Logaritmický graf času vyhledávání. Modře jsou vyznačeny běhy s vypnutým vylepšením. Tmavě modře a červeně pro DNA a světle modře a zeleně pro angličtinu.

vzorku v rozmezí 13–26 znaků. Hodnota σ tedy neklesala[33] pod IPM, ale byla značně snížena z původních 215 symbolů.

Ve všech případech měření pak bylo vyhledání se zapnutým vylepšením rychlejší. V přirozeném jazyce byl rozdíl velmi dobře patrný – vyhledávání bylo téměř desetkrát rychlejší. Důležité je také poznamenat, že počet nalezených výskytů nebyl zúžením abecedy v žádném případě ovlivněn.

5.2.4 Vliv metriky a zarovnávání

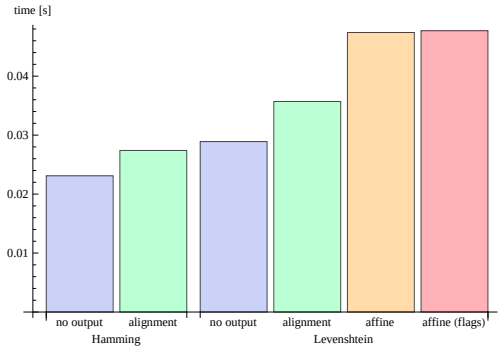
Následující experiment sleduje vliv zarovnávacího modulu na celkový čas vyhledávání. Dotaz a parametry vyhledávání proto byly nastaveny tak, aby byl tento vliv co nejpatrnější. Tedy relativně rychlé vyhledávání s poměrně velkým výstupem k zarovnání. Délka vzorku proto byla zvolena $|P| = 500$, $k = 5$, $s = 1$ a použito bylo zúžení abecedy.

Pokud by byl níže uvedený graf použit ke srovnání náročnosti vyhledávání v Hammingově a Levenshteinově vzdálenosti, je nutné doplnit, že pro Hammingovu vzdálenost existuje v textu jen 10 validních výskytů, zatímco pro Levenshteinovu 15.

Z grafu na obr. 5.5 lze odvodit, že pokud je 15 takto dlouhých sekvencí zarovnáno řádově za 10 ms, pak tato funkce uživatele nebude zdržovat. Graf potvrzuje, že afinní zarovnání je náročnější, avšak uživatelsky stále stejně příjemné. Poměrně zajímavým výsledkem je, že ukládání příznaku při výpočtu matic afinního zarovnání (na grafu červeně) je lehce pomalejší než opětovné výpočty příznaků při konstrukci konkrétního afinního zarovnání (oranžový sloupec).

K současné implementaci bych chtěl poznamenat, že zarovnání je vypsáno při prvním nálezu vzorku Filtrem. V editační vzdálenosti a při vyhledávání

[33] Výjimkou je jen $|P| = 20$, avšak připomeňme, že pokud by $|P|$ bylo rovno IPM, nesměl by se ve vzorku žádný znak opakovat, což je v přirozeném jazyce neobvyklé.



Obrázek 5.5: Graf celkového času vyhledávání. Modře je vyznačen čas samotného vyhledávání, tedy bez zarovnání a výpisu výskytů.

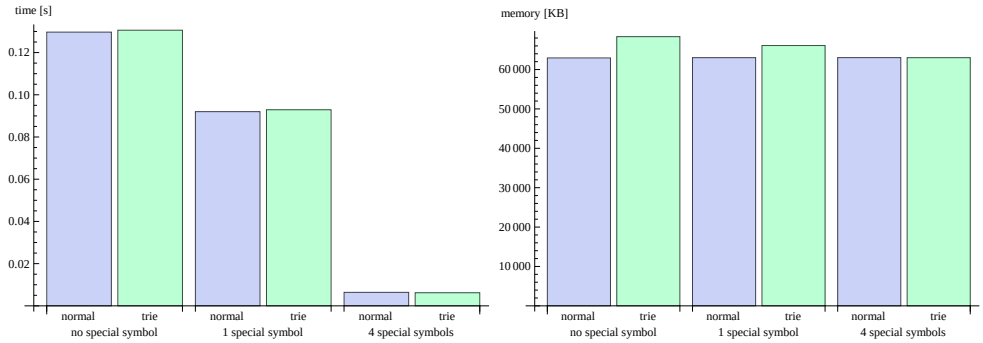
vzorku s chybami se pak může stát, že filtr interpretuje nejdříve chyby neoptimálním způsobem. Zarovnávací modul tuto chybu opraví. Ve výpisu je zobrazen i počet chyb stanovený Filtrem, lze tedy tuto situaci odlišit. Zarovnání se v tomto případě podílí na verifikaci, ale je na místě poznamenat, že i bez tohoto modulu je Filtr schopen stanovit správný počet chyb ve výskytu. Optimální konfigurace však může být řešena později a výpis by musel být odložen na závěr vyhledávání. Díky funkci optimálního zarovnání však pozdější konfigurace téhož výskytu nemusí být řešeny. V současné implementaci se stejně postupuje i s vypnutým zarovnávacím modulem, neboť nastavení je chápáno tak, že požadovaným výstupem je pouze počet validních výskytů.

Zmíněný problém může být řešen více způsoby než těmito, např. zajištěním, že nejdříve budou kontrolovány výskyty podvzorku s nezměněnou velikostí, nebo tzv. teleskopickou verifikací. Ve druhém případě by bylo provedeno více než o polovinu více verifikací. Zvolil jsem aktuální řešení kvůli jeho jednoduchosti, a nízké časové i paměťové náročnosti.

5.2.5 Vliv použití trie

V následujících experimentech je srovnáváno také využití trie, navržené v sekci 4.5.1. V obou experimentech byl maximální povolený poměr chyb $\alpha = 10\%$ a parametry algoritmu nastaveny na filtraci s vyhledáváním s jednou chybou. Byla použita metrika Levenshteinovy vzdálenosti. Každý z experimentů byl proveden s jinou délkou vzorku, z jejich kontrolních měření je tak možné vysledovat vliv délky vzorku na techniku využívající trie.

První experiment zkoumá vliv tzv. *speciálního symbolu*, jak byl popsán v sekci 4.5.2. Jedná se o symbol, jenž při indexování textu nemohl být pozorován. Vzorky délky $|P| = 40$ byly upraveny, tak že některé znaky byly přeměněny na takový symbol. V při pokusu s více speciálními symboly, byly přeměněny rovnoměrně vzdálené znaky. Jako speciální symbol jsem zvolil znak '?', jenž se v souboru *Escherichia_Coli* nevyskytuje.



Obrázek 5.6: Vliv přítomnosti speciálních symbolů ve vzorku (zeleně případ s použitím trie). Levý graf zobrazuje čas vyhledávání, pravý spotřebu operační paměti.

Při kontrolním měření byl vyhledán původní vzorek, dále bylo vyhledáváno s jedním speciálním symbolem respektive se čtyřmi. Jak je vidět na obrázku 5.6, znalost abecedy indexovaného textu může v tomto případě zásadně ovlivnit čas vyhledávání. Ze stejného grafu je zřejmé, že rozdíl času při použití trie byl zanedbatelný. Usuzuji, že zrychlení, které použití trie přineslo, se vyrovnalo režii nutné k vytvoření, procházení a zrušení datové struktury trie.

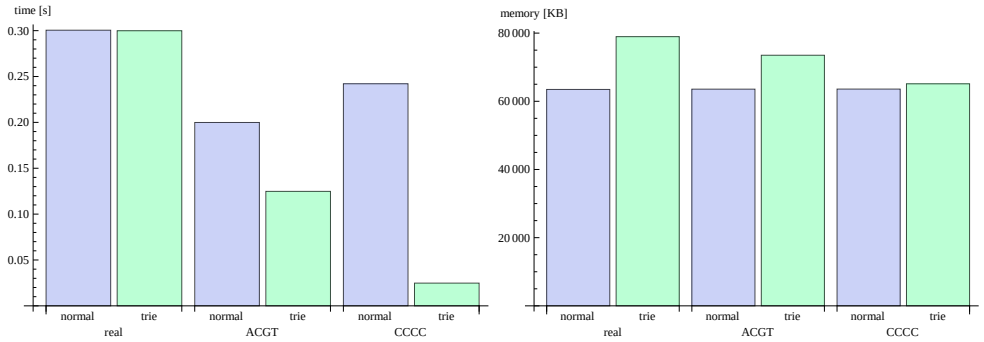
Graf zobrazující využití paměti ukazuje, že vyhledávání se speciálními symboly běžně neovlivňuje spotřebu paměti. Avšak při ukládání generovaných okolí podřetězců do trie mohou být tyto dodatečné paměťové nároky sníženy na minimum. Graf také dokazuje, že při tomto nastavení je i trie největší velikosti poměrně malý.

Aby byly na grafu rozdíly pozorovatelné, bylo z celkové spotřeby programu odečteno 110048 KB. Tento objem odpovídá velikosti prohledávaného textu. Před vyhledáváním modul indexu načte do operační paměti zapracovaný indexový soubor. Lze očekávat, že touto operací bude spotřebováno minimálně tolik paměti, kolik zabírá soubor na disku.

Stejným způsobem byly vytvořeny grafy i pro druhý experiment, jenž jsou vyobrazeny na obrázku 5.7. Ten se také zaměřuje na vliv obsahu vzorku, ale zkoumanou vlastností byla repetitivnost. Vyhledávány byly vždy vzorky délky $|P| = 100$. V prvním případě, jenž je kontrolním měřením, byl použit skutečný úsek DNA. V druhém případě byl repetitivní vzorek vytvořen opakováním řetězce „`ACGT`“. Poslední případ je v tomto smyslu extrémní, neboť vzorek je tvořen opakováním symbolu '`C`'.

Z grafu pro čas vyhledávání vyplývá, že tato vlastnost má nezanedbatelný vliv na oba přístupy. Přístup využívající trie se ukázal ve všech případech rychlejší. Pro více repetitivní vzorky je rozdíl větší (pro extrémní případ bylo využití trie desetkrát rychlejší).

Graf spotřeby paměti opět ukazuje, že pro standardní postup nebyla spotřeba obsahem vzorku ovlivněna, zatímco při použití trie klesala s repetitiv-



Obrázek 5.7: Vliv repetitivních vzorků na vyhledávání (zeleně případ s použitím trie). Levý graf zobrazuje čas vyhledávání, pravý spotřebu operační paměti.

ností vzorku velikost trie na minimum. V nejhorším případě pak tato datová struktura zabírala 15 MB, což považuji za únosné.

Experimenty ukázaly, že s délkou vzorku se časové i paměťové nároky zvyšují, ale přístup používající ukládání jednotlivých k-okolí do trie byl prakticky využitelný a pro repetitivní vzorky výhodný.

5.3 Srovnání s BLASTem

Svou implementaci s indexem FMIv2, jenž je vhodný pro textová data, jsem srovnával s BLASTem, jenž je současným běžným řešením pro vyhledávání v DNA. Porovnání bylo provedeno s posledním *BLAST Legacy* i s nejnovější verzí *BLAST+*. Tyto programy jsou implementací algoritmu popsaného v [35], jenž je vylepšením základní verze BLASTu, která byla popsána v sekci 2.4.

Programy byly použity se svým výchozím nastavením, ale snažil jsem se experimenty nastavit tak, aby programy řešily co nejpodobnější úlohu. Proto byly v zájmu dobrého srovnání některé parametry upraveny.

BLAST Legacy – použil jsem implementaci *NCBI-BLAST 2.2.26* [34]. Při vytváření databáze z FASTA souboru nástrojem *formatdb* bylo požadováno také parsování identifikátorů sekvencí a vytvoření indexu. Vysvětlení poskytuje sekce 5.3.1. Pro vyhledávání byl použit program *blastn* s jádrem *legacy BLAST*. Byl vypnut *DUST* filtr vzorku.

BLAST+ – použil jsem implementaci *NCBI-BLAST 2.2.30+* [35]. Při indexování FASTA souboru nástrojem *makeblastdb* bylo také vyžadováno parsování identifikátorů sekvencí. Při vyhledávání jsou identifikátory zobrazeny. Pro vyhledávání je vynucen algoritmus *blastn* a vypnuto filtrování vzorku programem *DUST*. Výstup byl nastaven tak, aby výpis výskytů nemusel být za-

[34] `ftp://ftp.ncbi.nlm.nih.gov/blast/executables/release/LATEST/`
[35] `ftp://ftp.ncbi.nlm.nih.gov/blast/executables/blast+/LATEST/`

lamován. Tato verze BLASTu je první, která umožňuje filtrovat výstup. Kritéria proto byla nastavena tak, aby byly vypisovány jen výskyty dostatečně podobné vzorku. Poměr požadované minimální shody je odvozen z hodnoty α, jenž vyplývá z maximálního počtu chyb k. Také je požadováno vyhledání celého vzorku.

APX_FILTER – takto je na grafech označena má implementace s FM-indexem, jenž je používán se svým výchozím nastavením. Vyhledávání je prováděno v editační vzdálenosti a zapnutým výstupem. Výskyty jsou zobrazeny v afinním zarovnání se vzorkem a je vypočítáno jeho skóre. Jako vyhledávací algoritmus bylo použito nastavení pro *Intermediate Partitioning* nastavený dle nerovnice 3.14, tedy tzv. filtr pro vyhledávání s jednou chybou. Vzhledem k důvodu vysvětlenému v následující sekci bylo použito navržené zúžení abecedy.

5.3.1 Použitá data

Aby testy mohly být prováděny všemi programy na stejných datech, bylo třeba vybrat takový soubor textu, jenž je BLASTem podporován. Typickým vstupem pro BLAST je databáze sekvencí ve formátu *FASTA*.

FASTA formát je založen na holých sekvencích DNA, tedy řetězci *IUPAC* symbolů, kde každý znak velkého písmena reprezentuje jednu bázi. Každých nejvýše 80 znaků sekvence by měl oddělovat znak konce řádku. Toto doporučení není standardizováno, ale je dodržováno. Jednotlivé sekvence jsou odděleny znakem '>' na začátku řádku. Do konce tohoto řádku je prostor pro anotaci následující sekvence a za tímto řádkem sekvence následuje [21].

Ačkoliv to není standardizováno, typicky anotace začíná tzv. *gi* identifikátorem, za nímž po mezeře následuje stručný popis v angličtině. NCBI používá k jednoznačné identifikaci sekvence identifikátor složený z řetězců oddělených znakem '|' [21]. Obsahem řetězců jsou většinou číslice. Identifikátor lze poznat podle toho, že první řetězec má hodnotu '`gi`'.

BLAST může při indexaci obsah oddělovacích řádků ignorovat, případně uložit jen identifikátor. Dále jsou vynechány znaky pro zalamování sekvencí. Výhodné by také bylo nahradit všechny nejednoznačné báze jedním symbolem. S výhodou lze také využít znalosti hranic jednotlivých sekvencí v textu. Protože výskyt nemůže být lokalizován na rozhraní sekvencí, lze tyto potenciální výskyty rovnou vyloučit a zvýšit tak filtrační výkon. Tyto návrhy jsem nerealizoval, protože implementovaný algoritmus je obecný a nespecializuje se na DNA. Nicméně v implementaci BLASTu lze podobná vylepšení očekávat.

Pro experimenty byl jako text zvolen soubor *human_genomic*, přesněji jeho prefixy různých délek. Data jsem získal z NCBI[36]. Jedná se o referenční sekvenci lidského genomu ve FASTA formátu. Soubor byl vybrán, protože ob-

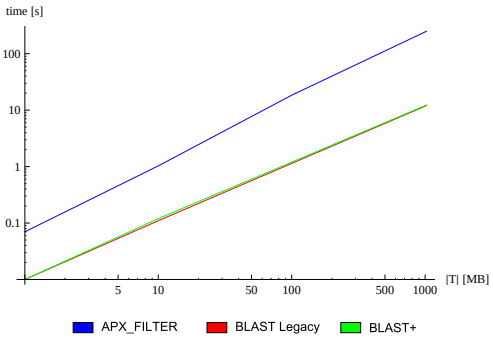
[36] `ftp://ftp.ncbi.nlm.nih.gov/blast/db/FASTA/`

sahuje celé chromozomy, nikoliv mnoho krátkých sekvencí. Tím jsou omezeny některé výhody specializovaného řešení oproti obecnému.

Anotace, jenž nejsou předmětem vyhledávání, však zvyšují velikost abecedy textu. V tomto případě na $\sigma = 40$. Aby byla minimalizována výhoda BLASTu založená na vynechávání anotačních řádků z vyhledávání, použil jsem tzv. zúžení abecedy. Realizovaný program proto automaticky vyhledával nad abecedou vyhledávané DNA sekvence, jenž má $\sigma = 4$.

5.3.2 Vytváření indexového souboru

V následujícím experimentu je sledováno indexování vstupního textu v závislosti na délce textu. Použita byla reálná DNA o délkách 1 MB, 10 MB, 100 MB a 1 GB. Z tohoto důvodu mají vytvořené grafy logaritmické měřítko na obou osách.



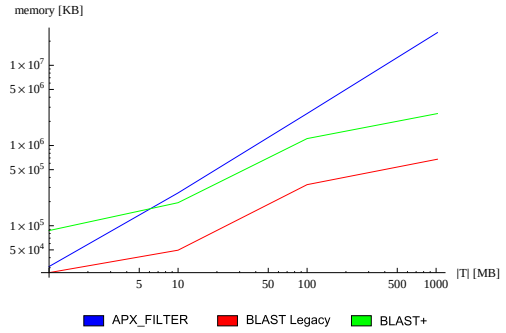
Obrázek 5.8: Vliv velikosti vstupního textu na čas potřebný k vytvoření indexového souboru.

Z grafu na obrázku 5.8 je patrné že implementace FMIv2, kterou jsem použil ve svém programu, je při vytváření indexu o řád pomalejší než BLAST. Obě verze BLASTu jsou srovnatelné, nicméně z grafu není příliš zřejmá skutečnost že BLAST Legacy je rychlejší.

Maximální spotřeba operační paměti je na grafu na obr. 5.9. FMIv2 má lineární chování a během měření spotřeboval paměť zhruba 25-krát větší než indexovaný soubor. Tato implementace se proto pro velké soubory ukazuje nevhodnou. Ve zkoumaných případech má jednoznačně nejnižší spotřebu BLAST Legacy, ale graf také ukazuje, že pro menší soubory FMIv2 předčí BLAST+ a je možné, že pro texty v řádu KB předčí i BLAST Legacy.

5.3.3 Vliv velikosti textu

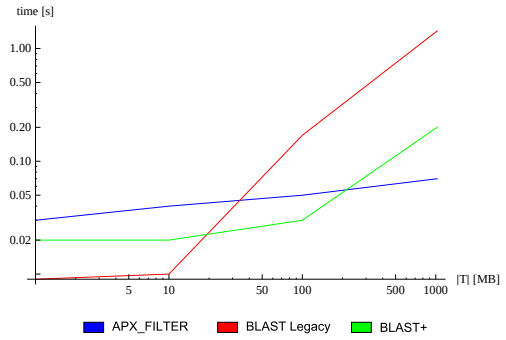
Tento experiment zkoumá vliv velikosti prohledávané DNA sekvence na samotné vyhledávání. Použity byly texty indexované v předchozím experimentu (viz sekce 5.3.2). Zmiňovanou výhodou specializovaného řešení je vynechávání



Obrázek 5.9: Vliv velikosti vstupního textu na spotřebu paměti potřebné k vytvoření indexového souboru.

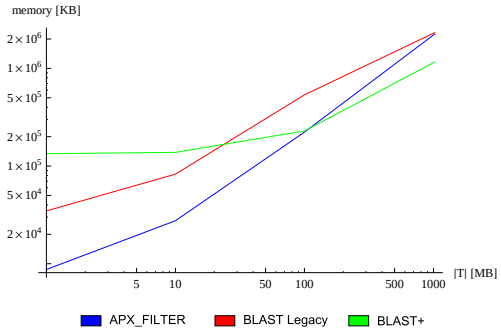
některých částí textu i vzorku (např. rovnoměrně rozmístěné znaky ukončující řádek). Proto při prohledávání například 1 MB souboru program APX_FILTER prohledával 1048576 znaků, zatímco BLAST jen 1035547 znaků.

Vyhledáván byl vzorek délky $|P| = 75$, tato délka byla vybrána, protože vzorek, ani jeho výskyt v textu, nemusí obsahovat znak konce řádku. Vyhledáváno bylo s $\alpha = 4\%$, což odpovídá přibližnému vyhledávání s nejvýše $k = 3$ chybami v Levenshteinově vzdálenosti.



Obrázek 5.10: Logaritmický graf vlivu velikosti vstupního textu na čas vyhledávání.

APX_FILTER má čas vyhledávání logaritmický vzhledem k velikosti textu. Z grafu na obrázku 5.10 lze vypozorovat, že chování BLASTu je odlišné. Pro krátké soubory se časy o mnoho neliší, ale od určité velikosti textu poměrně rychle roste. BLAST Legacy je pro malé soubory nejrychlejší, ale pro velké nejpomalejší. APX_FILTER je pro velké soubory jednoznačně nejlepší. Připomínám, že osy grafu jsou v logaritmickém měřítku, proto rozdíl mezi krátkými časy je zobrazován větší a rozdíl mezi dlouhými časy je zobrazován menší. Například pro 1 GB textu je v tomto nastavení APX_FILTER téměř třikrát rychlejší než BLAST+ a více než 20-krát rychlejší než BLAST Legacy.



Obrázek 5.11: Logaritmický graf vlivu velikosti vstupního textu na maximum využité operační paměti během vyhledávání.

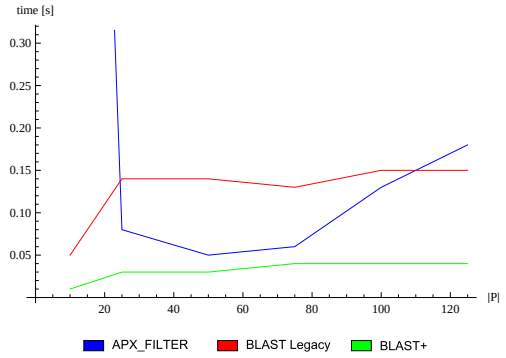
Graf na obrázku 5.11 ukazuje maximální spotřebu paměti programů. Ta je spíše než algoritmy pro vyhledávání ovlivněna indexovacími algoritmy, které prohledávaný text v určité podobě ukládají do operační paměti. BLAST+ má pro menší soubory výsledky nejhorší, zatímco pro opravdu velké soubory má spotřebu nejnižší. Kvůli logaritmickému měřítku není z grafu patrné, že pro 1 GB dlouhý text je nejhorší BLAST Legacy, který spotřeboval o více než 76 MB více paměti než APX_FILTER. APX_FILTER je pak nejlepším pro relativně širokou škálu menších textů (pro 100 MB dlouhý text spotřeboval o 6 MB méně než BLAST+). Z grafu je patrné že pro 1 MB dlouhý text jsou mezi programy rozdíly ve spotřebě dokonce řádové.

Z výsledků měření jsem zjistil že APX_FILTER při prohledávání 1 GB dlouhého souboru nenalezl jeden výskyt vzorku, jenž v ostatních případech nalezl. Proto se domnívám, že chyba pravděpodobně byla způsobena v modulu FMIv2, jehož použití zřejmě autory nebylo očekáváno na tak velkých textech.

5.3.4 Vliv délky vzorku

V tomto experimentu je prohledávána DNA o velikosti 100 MB, vždy s $\alpha = 10\%$ v editační vzdálenosti. Měněna je jen délka vyhledávaného vzorku. Použity byly hodnoty $|P| = 10$ a 25–125 s krokem 25. Osy grafů jsou proto v lineárním měřítku.

Graf na obrázku 5.12 ukazuje že BLAST má časy pro $|P| > 10$ relativně neměnné. Dá se říci že BLAST+ je rychlejší než BLAST Legacy a APX_FILTER se svými výsledky řadí mezi tyto verze BLASTu. Pro APX_FILTER však na grafu můžeme pozorovat zhoršení, když délka vzorku přesáhne 80. Tuto skutečnost lze vysvětlit tím, že prohledávaný FASTA soubor má definovanou délku řádku na 80 znaků. Dlouhé vzorky proto obsahují znak zalomení řádku. BLAST tento znak ignoruje, zatímco obecný algoritmus jej započítává do abecedy. Při nastavení v jakém byl APX_FILTER spuštěn pak má toto falešné rozšíření abecedy DNA sekvence nezanedbatelný negativní vliv na rychlost



Obrázek 5.12: Vliv velikosti vzorku na čas vyhledávání.

vyhledávání.

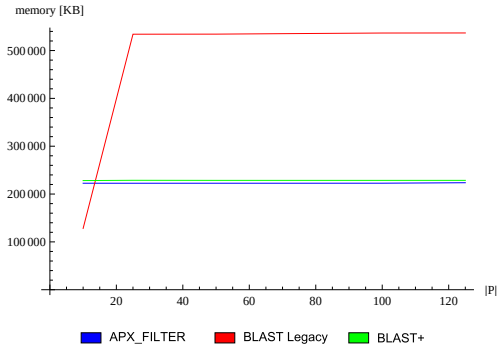
Další zajímavá situace je $|P| = 10$. Pro takto krátké vzorky BLAST zcela selhává. Obě verze hlásí, že v textu nebyl nalezen žádný výskyt. Předčasné ukončení programů lze na grafu pozorovat. Naopak poměrně dlouhý čas vyhledávání programu APX_FILTER je zapříčiněn vysokým počtem nalezených výskytů. Program vypsal informace o 9429 nalezených výskytech za 1,73 s (v průměru každý výskyt za 0,18 ms). Vyhledávání bych proto označil za velmi rychlé. Filtr byl v tomto případě nastaven v režimu „Neighborhood Generation“, což ukazuje úspěšné uplatnění tohoto filtračního přístupu (viz sekce 2.3.1).

Porovnávání počtu nalezených výskytů je však v některých případech problematické. Například BLAST Legacy generuje poměrně velké výstupy – řádově větší než APX_FILTER. Důvodem je to, že neumožňuje stanovit maximální tolerovanou chybu při přibližném vyhledávání, ani určit že vyhledáván má být celý vzorek. Ve výstupu programu je tak mnoho nevalidních výskytů. Poslední[37] verze BLAST+ již tato kritéria umožňuje nastavit, proto jsem počet nalezených výskytů srovnával s touto verzí BLASTu.

Dalším problémem je výše zmíněný fakt, že v použitém textu sekvence DNA obsahují zalamování řádků. Pokud není sekvence výskytu zalomena stejným způsobem jako sekvence vzorku, pak musí obecný algoritmus každý znak konce řádku považovat za chybu. Tyto falešné chyby mohou zapříčinit, že výskyt bude posouzen jako nevalidní, zatímco pokud by sekvence zalamovány nebyly, validní by byl. Proto lze jako signalizaci chyby použít jen situaci kdy BLAST reportuje méně výskytů než APX_FILTER. Tato situace nastala, kromě případu $|P| = 10$, také pro $|P| = 25$. APX_FILTER lokalizoval 12 výskytů, zatímco BLAST+ jen 5.

Graf na obrázku 5.13 vyobrazuje celkovou spotřebu paměti jednotlivých programů. BLAST Legacy má více než dvojnásobnou spotřebu než zbývající programy. Výjimkou je jen téměř poloviční spotřeba v případě, kdy vyhledá-

[37] Nastavení obou kritérií umožnila verze z října 2014.[36]



Obrázek 5.13: Vliv velikosti vzorku na maximum využité operační paměti během vyhledávání.

vání prakticky nebylo provedeno. Nejlepších výsledků dosahuje APX_FILTER (spotřeba je v této situaci zhruba o 6 MB nižší než u BLAST+). Spotřeba paměti sice s velikostí vzorku roste, ale jak z grafu vyplývá, zanedbatelným způsobem.

Závěr

Během rešerše jsem se seznámil s několika přístupy pro přibližné vyhledávání v textu. Zvláště jsem se zaměřil na tzv. filtrační algoritmy, jenž s výhodou využívají indexování. Navrhl jsem pak poměrně obecný filtrační algoritmus založený na *pigeonhole* principu.

Návrh je založený na modulární struktuře propojující různé algoritmy. Byly implementovány algoritmy pro filtraci, verifikaci, globální zarovnání a generování okolí řetězce. Jako vlastní index jsem použil existující implementaci FM-Indexu, *FMIv2*. Každý z těchto algoritmů může být nahrazen jiným a program tak může být vylepšen či lépe specializován pro určitou aplikaci.

Program byl úspěšně realizován a je schopen přibližného vyhledávání v textu s použitím Hammingovy i Levenshteinovy vzdálenosti. Program je možné přizpůsobit i pro jiné metriky. V případě složitější definice chyby je však nutné odpovídajícím způsobem upravit i algoritmus filtrace.

Program je prakticky použitelný, ale nelze zaručit jeho 100% bezchybnost. Jedním důvodem byl nedostatek času pro extenzivní testování a druhým je skutečnost, že takovou spolehlivost nezaručují ani převzaté kódy FM-Indexu. V implementaci FMIv2 jsem však řadu chyb odhalil a úspěšně opravil.

V budoucí práci doporučuji změnit implementaci indexu a opravit objevené chyby. Současný indexovací algoritmus není vhodný pro binární data, či zpracování textů o délce v řádu GB. K filtračnímu algoritmu, který jsem realizoval se tyto problémy nevztahující.

Ve své práci jsem také navrhl základní postupy a vzorce pro automatické nastavení parametrů vyhledávání. Prostor parametrů problému i algoritmu je poměrně složitý, předmětem další práce by mohl být algoritmus pro určení nejlepšího nastavení programu pro konkrétní vyhledávací dotaz.

Navržena byla také různá vylepšení. Účinnost realizovaných vylepšení v daných situacích byla prokázána experimenty. Program je použitelný na různé druhy textů, ale na poli prohledávání DNA byl srovnán se specializovaným programem BLAST. Měření pak ukázalo, že v některých aspektech je má implementace lepší než nejnovější verze BLASTu.

Funkce programu byla úspěšně otestována na platformě *Windows* i *GNU/Linux*, na 32b i 64b architektuře.

Literatura

[1] Moran, L. A.: How Big Is the Human Genome? In *Sandwalk*, Březen 2011, [cit. 2015-04]. Dostupné z: `http://sandwalk.blogspot.cz/2011/03/how-big-is-human-genome.html`

[2] Pray, L. A.: DNA replication and causes of mutation. *Nature Education*, ročník 1, č. 1, 2008: str. 214. Dostupné z: `http://www.nature.com/scitable/topicpage/dna-replication-and-causes-of-mutation-409`

[3] Ferragina, P.; Venturini, R.: *FM-Index Version 2*. Dipartimento di Informatica, University of Pisa, Září 2005, [cit. 2015-04]. Dostupné z: `http://www.di.unipi.it/~ferragin/Libraries/fmindexV2/`

[4] Navarro, G.: A Guided Tour to Approximate String Matching. *ACM Comput. Surv.*, ročník 33, č. 1, Březen 2001: s. 31–88, ISSN 0360-0300, doi:10.1145/375360.375365. Dostupné z: `http://doi.acm.org/10.1145/375360.375365`

[5] Ukkonen, E.: Algorithms for approximate string matching. *Information and Control*, ročník 64, č. 1–3, 1985: s. 100–118, ISSN 0019-9958, doi: http://dx.doi.org/10.1016/S0019-9958(85)80046-2, international Conference on Foundations of Computation Theory. Dostupné z: `http://www.sciencedirect.com/science/article/pii/S0019995885800462`

[6] Hamming, R.: Error detecting and error correcting codes. *Bell System Technical Journal, The*, ročník 29, č. 2, Duben 1950: s. 147–160, ISSN 0005-8580, doi:10.1002/j.1538-7305.1950.tb00463.x.

[7] Lee, C.: Some properties of nonbinary error-correcting codes. *Information Theory, IRE Transactions on*, ročník 4, č. 2, Červen 1958: s. 77–82, ISSN 0096-1000, doi:10.1109/TIT.1958.1057446.

[8] Levenshtein, V. I.: Binary codes capable of correcting deletions, insertions, and reversals. *Doklady Akademii Nauk SSSR*, ročník 163, č. 4, 1965: str. 845–848.

[9] Damerau, F. J.: A Technique for Computer Detection and Correction of Spelling Errors. *Commun. ACM*, ročník 7, č. 3, Březen 1964: s. 171–176, ISSN 0001-0782, doi:10.1145/363958.363994. Dostupné z: http://doi.acm.org/10.1145/363958.363994

[10] Needleman, S. B.; Wunsch, C. D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, ročník 48, č. 3, 1970: s. 443–453, ISSN 0022-2836, doi:http://dx.doi.org/10.1016/0022-2836(70)90057-4. Dostupné z: http://www.sciencedirect.com/science/article/pii/0022283670900574

[11] Smith, T.; Waterman, M.: Identification of common molecular subsequences. *Journal of Molecular Biology*, ročník 147, č. 1, 1981: s. 195–197, ISSN 0022-2836, doi:http://dx.doi.org/10.1016/0022-2836(81)90087-5. Dostupné z: http://www.sciencedirect.com/science/article/pii/0022283681900875

[12] Polyanovsky, V.; Roytberg, M.; Tumanyan, V.: Comparative analysis of the quality of a global algorithm and a local algorithm for alignment of two sequences. *Algorithms for Molecular Biology*, ročník 6, č. 1, 2011: str. 25, ISSN 1748-7188, doi:10.1186/1748-7188-6-25. Dostupné z: http://www.almob.org/content/6/1/25

[13] Navarro, G.; Baeza-yates, R.; Sutinen, E.; aj.: Indexing Methods for Approximate String Matching. *IEEE Data Engineering Bulletin*, ročník 24, č. 4, 2001: s. 19–27. Dostupné z: http://www.sigmod.org/publications/ds-collection/discs-new/2002/out/websites/deb/a01dec/A01DEC-CD.pdf#page=21

[14] Navarro, G.; Baeza-Yates, R.: A Hybrid Indexing Method for Approximate String Matching. *J. of Discrete Algorithms*, ročník 1, č. 1, 2000: str. 205–239. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.134.9634

[15] Altschul, S. F.; Gish, W.; Miller, W.; aj.: Basic local alignment search tool. *Journal of Molecular Biology*, ročník 215, č. 3, 1990: s. 403–410, ISSN 0022-2836, doi:http://dx.doi.org/10.1016/S0022-2836(05)80360-2. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0022283605803602

[16] Advanced Biocomputing, LLC: *Frequently Asked questions*. 2014, [cit. 2015-04]. Dostupné z: http://www.advbiocomp.com/faq.html

[17] Camacho, C.; Coulouris, G.; Avagyan, V.; aj.: BLAST+: architecture and applications. *BMC Bioinformatics*, ročník 10, č. 1, 2009: str. 421, ISSN 1471-2105, doi:10.1186/1471-2105-10-421. Dostupné z: `http://www.biomedcentral.com/1471-2105/10/421`

[18] Bethesda (MD): National Center for Biotechnology Information (US): *BLAST® Command Line Applications User Manual*. 2008, [cit. 2015-04]. Dostupné z: `http://www.ncbi.nlm.nih.gov/books/NBK279690/`

[19] NCBI: *Blast Program Selection Guide*. Červenec 2009, [cit. 2015-04]. Dostupné z: `http://blast.ncbi.nlm.nih.gov/BLAST_guide.pdf`

[20] NCBI: *BLAST Frequently Asked questions*. [cit. 2015-04]. Dostupné z: `http://blast.ncbi.nlm.nih.gov/Blast.cgi?CMD=Web&PAGE_TYPE=BlastDocs&DOC_TYPE=FAQ`

[21] NCBI: *Query Input and database selection*. [cit. 2015-04]. Dostupné z: `http://blast.ncbi.nlm.nih.gov/blastcgihelp.shtml`

[22] Computational Biology Research Group: *BLAST filtering*. Březen 2005, [cit. 2015-04]. Dostupné z: `http://www.compbio.ox.ac.uk/analysis_tools/BLAST/BLAST_filtering.shtml`

[23] Morgulis, A.; Gertz, E. M.; Schäffer, A. A.; aj.: A Fast and Symmetric DUST Implementation to Mask Low-Complexity DNA Sequences. *Journal of Computational Biology*, ročník 13, č. 5, Červen 2006: s. 1028–1040, doi:10.1089/cmb.2006.13.1028. Dostupné z: `http://online.liebertpub.com/doi/abs/10.1089/cmb.2006.13.1028`

[24] Wootton, J. C.; Federhen, S.: Analysis of compositionally biased regions in sequence databases. In *Computer Methods for Macromolecular Sequence Analysis*, *Methods in Enzymology*, ročník 266, editace R. F. Doolittle, Academic Press, 1996, s. 554–571, doi:http://dx.doi.org/10.1016/S0076-6879(96)66035-2. Dostupné z: `http://www.sciencedirect.com/science/article/pii/S0076687996660352`

[25] Myers, G.: What's behind BLAST. In *Combinatorial Pattern Matching, 25th Annual Symposium, CPM 2014: Moscow, Russia*, 2014, invited talk. Dostupné z: `http://www.cs.ucr.edu/~stelo/cpm/cpm2014.html`

[26] Burkhardt, S.: *Filter algorithms for approximate string matching*. Dizertační práce, Universität des Saarlandes, Postfach 151141, 66041 Saarbrücken, 2002. Dostupné z: `http://scidok.sulb.uni-saarland.de/volltexte/2004/171`

[27] Miller, J.; Flor, P.; Berg, G.; aj.: *Earliest Known Uses of Some of the Words of Mathematics*. [cit. 2015-04]. Dostupné z: `http://jeff560.tripod.com/mathword.html`

[28] Baeza-Yates, R.; Navarro, G.: Multiple approximate string matching. In *Algorithms and Data Structures*, *Lecture Notes in Computer Science*, ročník 1272, editace F. Dehne; A. Rau-Chaplin; J.-R. Sack; R. Tamassia, Springer Berlin Heidelberg, 1997, ISBN 978-3-540-63307-5, s. 174–184, doi:10.1007/3-540-63307-3_57. Dostupné z: http://dx.doi.org/10.1007/3-540-63307-3_57

[29] Kawulok, J.: Approximate String Matching for Searching DNA Sequences. *International Journal of Bioscience, Biochemistry and Bioinformatics*, ročník 3, č. 2, Březen 2013: s. 145–148. Dostupné z: http://www.ijbbb.org/show-36-445-1.html

[30] Altschul, S. F.: *The Statistics of Sequence Similarity Scores*. NCBI, [cit. 2015-04]. Dostupné z: http://www.ncbi.nlm.nih.gov/blast/tutorial/Altschul-1.html

[31] Weisstein, E. W.: Lambert W-Function. In *MathWorld*, A Wolfram Web Resource, [cit. 2015-04]. Dostupné z: http://mathworld.wolfram.com/LambertW-Function.html

[32] Eveleigh, V.: Bioinformatics algorithms. Jesus College, Cambridge University. Dostupné z: http://www.vaughaneveleigh.co.uk/academic/bioinformatics-algorithms/

[33] Ferragina, P.; Navarro, G.: *A Repetitive Corpus Testbed*. Září 2005, [cit. 2015-04]. Dostupné z: http://pizzachili.dcc.uchile.cl/repcorpus.html

[34] Ferragina, P.; Navarro, G.: *Pizza&Chili Corpus*. Září 2005, [cit. 2015-04]. Dostupné z: http://pizzachili.dcc.uchile.cl/index.html

[35] Altschul, S. F.; Madden, T. L.; Schäffer, A. A.; aj.: Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, ročník 25, č. 17, 1997: s. 3389–3402, doi:10.1093/nar/25.17.3389. Dostupné z: http://nar.oxfordjournals.org/content/25/17/3389

[36] Camacho, C.: BLAST+ Release Notes. In *BLAST® Help*, Bethesda (MD): National Center for Biotechnology Information (US), Říjen 2014. Dostupné z: http://www.ncbi.nlm.nih.gov/books/NBK131777/

Příloha **A**

Seznam použitých zkratek

BASH Bourne again shell

BLAST Basic Local Alignment Search Tool

bp Base-pair (komplementární pár bází)

DFA Deterministický konečný automat

DNA Kyselina deoxyribonukleová

DP Dynamické programování

FASTA FAST-All (rychlé zarovnávání nezávislé na abecedě)

FMI FM-index (Ferragina-Manzini)

FMIv2 FM-index verze 2

gi GenInfo Identifier

GiB Gibibyte (2^{30} bajtů)

GNU „GNU's Not Unix!“ (UNIX-like OS)

GPU Graphics processor unit

IA-32 Intel Architecture, 32-bit

IPM Inverse of the Probability that two characters chosen at random Match

IUPAC International Union of Pure and Applied Chemistry

JEDEC Joint Electron Device Engineering Council

LF Last-to-first (zobrazení znaků posledního sloupce na první)

NCBI National Center for Biotechnology Information

NFA Nedeterministický konečný automat

NP Nedeterministicky polynomiální

OS Operační systém

PC Osobní počítač

PERL Practical Extraction and Reporting Language

PSI Position-Specific Iterated (BLAST)

RNA Kyselina ribonukleová

XOR Exclusive-or (výhradní disjunkce)

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... popis obsahu CD a postup pro rekonstrukci experimentů
- `exe` ..... adresář se spustitelnou formou implementace
  - `ia-32` ..... pro Windows 32b
    - `apx_filter_silent.exe` verze bez přidaných výpisů a měření času
    - `apx_filter_std.exe` .... verze s přidanými výpisy a měřením času
  - `x86-64` ..... pro Linux 64b
    - `apx_filter_std` ..... verze s přidanými výpisy a měřením času
    - `apx_filter_time` ..... verze pro důkladné měření času opakováním dotazu (počet opakování lze řídit prametrem `-N`)
- `src` ..... zdrojové kódy implementace
- `test` ..... soubory pro jednoduché otestování programu
  - `experimenty.rar` ..... data a skripty použité v experimentech
  - `data.zip` ..... výstupní data všech experimentů
  - `Escherichia_Coli` ..... příklad původního textu DNA sekvence
  - `Escherichia_Coli.fmi` ... komprimovaná sekvence – indexový soubor pro realizovaný program
  - `Ecoli_500.txt` ..... příklad vzorku DNA (prefix původní sekvence)
- `text` ..... text práce
  - `topic.pdf` ..... zadání práce ve formátu PDF
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `src` ..... zdrojová forma práce ve formátu LaTeX
    - `obr` ..... vložené obrázky
    - `bibliografie.bib` ..... citované zdroje
    - `DP_Hrbek_Lukáš_2015.pdf`
    - `DP_Hrbek_Lukáš_2015.tex`

Příloha C

Instalační příručka

Zdrojové kódy programu APX_FILTER jsou přiloženy na CD v adresáři `src`. Pro kompilaci projektu v Unixovém prostředí stačí v této složce spustit příkaz `make`.

Spustitelný soubor pak bude vytvořen jako `./dist/Release/Linux/apx_filter`. Implicitní nastavení ve zdrojovém kódu odpovídá verzi poskytující informační výpisy včetně měření doby běhu.

Složku je také možné otevřít jako projekt pro *NetBeans*. Součástí projektu je také kompilační konfigurace „Debug“ pro systém Windows (např. s *MinGW*).

Po spuštění programu bez parametrů je zobrazena stručná nápověda. Podrobnější manuál lze zobrazit spouštěním s parametrem `-help`.