Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Bakalářská práce

Algoritmy vyhledávání v řetězcích pro DNA aplikace

Plzeň, 2011 Ondřej Žďárský

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne *23.4.2011*

Algorithms for searching in strings for DNA aplications

This thesis contains basic information about DNA, about processing of DNA strings, especially about optimalization of PCR processes and the list of the seven most related algorithms for exact string matching in DNA. Mentioned algorithms have been tested against a real string (in this case part of the code left in DNA by jaundice) and measured times are presented in graphs. Moreover, the area of use has been indicated for each algorithm, so the whole process can score the best time efficiency possible.

Time and space complexity of each algorithm is mentioned by implication.

Finally, key features of each algorithm is presented in a table in the chapter Conclusion (Závěr).

Obsah

1. Úvod

Od položení základního kamene v roce 1869 Johannem Gregorem Mendelem (formulace základních genetických zákonů) až po zveřejnění kompletní lidské DNA sekvence v roce 2001 prošla genetika dynamickým vývojem a v současnosti ji lze označit za klíčový obor nejen v biologii, ale i v její nejpraktičtější aplikaci – zdravotnictví (potažmo také v kriminalistice a historii). Daleko od pravdy není konstatování, že do jisté míry je jejím poznáním ovlivněn každý z nás.

Specializovaným odvětvím genetiky je molekulární genetika, zaměřená na operace a čtení DNA nebo RNA sekvence (souborně: sekvence nukleových kyselin). Technologie molekulární genetiky jsou dnes schopny přečíst kompletní sekvenci každého živého organizmu (případně i lidského individua). Například u lidí se však tato sekvence skládá z obrovského množství informací (přesněji 3,3 miliard nukleotidových bází), přičemž každý lidský jedinec obsahuje ve svém genomu zcela unikátní DNA sekvence, odlišné dokonce i od svých rodičů. Z toho je vidět, že práce s genetickou informací je velice obtížná, ale přesto velmi žádoucí.

Cílem této práce je především zmapování problematiky zpracování genetické informace a její propojení s možnostmi výpočetní technologie. Vzhledem ke způsobu záznamu genetické informace půjde především o analýzu různých způsobů prohledávání textu s ohledem na vlastnosti DNA řetězců a nástinu jejich použití pro reálné aplikace dnešní genetiky.

2. DNA

Lidská DNA [1],[2] je v každé buňce zapsána do 46 nezávislých řetězů, 23 pochází ze spermie otce a 23 z vajíčka matky. Kvalitativně DNA obsahuje jak sekvence, které nejspíše nemají vůbec žádnou funkci (sobeckou DNA), tak i sekvence genové. Genové sekvence jsou pravými genetickými informacemi, neboť v nich jsou zakódovány informace pro tvorbu proteinů nebo i přímé výkonné informace (účinkující i bez překladu do proteinů).

Samotný gen však je pouze jakousi jednotkou dědičnosti, neboť je to jen jakási sekvence příkazů v DNA potřebná k vytvoření proteinu nebo vlastní funkční RNA. Je to právě až protein nebo RNA, které zajišťují funkcionalitu buňky a tím její anatomické určení nebo druhovou příslušnost. Například lidé a pes mají jak mnoho proteinů odlišných, tudíž i jiných buněk i konečné architektury těla, ovšem není od věci si uvědomit, že stejně tak budou mít oba mnoho proteinů společných, tj. téměř (ale nikdy ne úplně) identických (namátkově třeba proteiny pro tvorbu krve, protein pro nervovou soustavu, atd.).

Bylo již uvedeno, že základ DNA sekvence tvoří pořadí nukleotidových bází, respektive ještě detailněji bází purinového základu (zastoupené Adeninem a Guaninem - zkráceně **A** a **G**) anebo bází pyrimidinového typu (Thymin a Cytosin - báze **T** a **C**).

Vyšší jednotkou než je sled nukleotidových bází je tzv. genetický kód [3], to znamená zápis po trojicích nukleotidů v genech překládaných do proteinů. Právě genetický kód je totiž využit pro definici sekvence proteinů a to tak, že trojice nukleotidů – například sled tři adenosinů v DNA je posléze rozpoznán jako instrukce k přidání lysinu (jedna z dvaceti možných aminokyselin – stavebních kamenů proteinů). Jelikož je však proteinová detailní sekvence aminokyselin vždy definovaná i sekvencí DNA, soustřeďuji se zde jen na sekvence nukleotidové.

Konečně, evolucí se lidská DNA vyvinula jako chemicky velmi stabilní útvar – dvojpolymer dvou opačně orientovaných vláken, navzájem komplementárních tak, že purinové báze odpovídají bázím pyrimidinovým a naopak. Komplementarita je vytvářena párovaným vodíkovými můstky a takto spojenému páru nukleotidů se říká nukleotidový pár [4] a je jednotkou délky DNA. Na obou vláknech je tudíž uložena tatáž informace, pouze v jakémsi „negativu“ vůči protějšímu řetězci.

Vzhledem k rozdělení DNA do dvou prakticky identických vláken, pro záznam DNA informace stačí číst pouze z vlákna jednoho. Levá strana vlákna se vždy označuje jako 5´konec a pravá jako 3´konec. Nikoliv nezbytně nutným, ale přesto dodržovaným zvykem je kvůli lepší čitelnosti záznam kódu rozdělit do sekvencí po šestkrát deseti znacích na řádek, navíc na začátku každého řádku je číslo značící pořadí prvního znaku z řádku v celkovém řetězci. DNA záznam může vypadat následovně:

```
1  acgcctatgc gtttacaaca cggatcggat ttgtggaaat cggctaatcg actgtgctat
61 gtttagtaca acgaggctgc ctgtgggaat cagatcggat catccaatcg gctgtgctat
```

Pro úplnost je dobré vést v patrnosti i to, že kromě již zmiňovaných zkratek nukleotidových bází (A, C, G, T) [5] se v praxi mohou vyskytnout v zápisu i některá další písmena. Tato písmena signalizují, že sekvence v tomto místě může obsahovat i více možností. Jde o zkratku, kdy se současně popisují obě rodičovská vlákna, kterážto nemusí být úplně totožná. Jde o znaky **K** (pro G a T), **M** (pro A a C), **W** (pro A a T), **S** (pro C a G), **Y** (pro C a T), **R** (pro A a G), **H** (A,C,T), **V** (A,C,G), **B** (C,G,T), **D** (A,G,T). V praxi se také využívá obecného znaku **N**, kterým se zastupuje libovolnou jednoznačně nerozpoznanou báze.

3. Zpracování DNA záznamů

Jak již bylo uvedeno výše, genetika ovlivňuje mnoho různých odvětví, přičemž každé odvětví má jiné cíle a požadavky. Z tohoto důvodu ani zpracování DNA není stejné a existuje hned několik různých způsobů zpracování jednoho záznamu [6].

- Vyhledávání restrikčních míst restrikčních endonukleáz – Restrikční endonukleázy jsou části genetického kódu, které chrání bakterie před napadením viry. V sedmdesátých letech minulého století umožnil objev restrikčních endonukleáz a jejich specifických rozpoznávacích míst v DNA, rozvoj biotechnologie.

- Srovnávání individuálních genomů s cílem nálezu mutací (míst s odlišnou sekvencí)

- Evoluční výpočet podle stupně rozdílu mezidruhových genomů – podle procenta shodného DNA lze určit celé vývojové větve jednoho druhu.

- Optimalizace PCR procesu – Zkratka PCR vznikla z anglického Polymerase Chain Reaction a jejím smyslem je namnožení vybraného krátkého úseku DNA. Takto namnožený, jinými slovy naklonovaný vzorek je možné otestovat na mnoho dědičných chorob, zjistit rodičovství dvou vzorků, nebo i pomoci odhalit pachatele trestných činů. Z šíře svého použití lze říci, že se jedná o hlavní výzkumnou technologii molekulární genetiky.

3.1. Optimalizace PCR procesu

Z důvodu široké využitelnosti PCR technologie v biologii i zdravotnictví jsem se zaměřil v této práci na tento druh zpracování genetické informace. PCR je metodologie, umožňující namnožení vybrané sekvence do takového množství, že je pak snadno analyzovatelná navazujícími metodami. Smyslem optimalizace PCR procesu je identifikování a výběr vhodných oligonukleotidů – několikabázových sekvencí DNA, kteržto ohraničují cílovou sekvenci z obou stran. Tento oligonukeotidový pár je definován pomocí několika pravidel [6]:

1. Pravidlo komplementarity - Nukleotidové řetězce mohou být jen částečně komplementární. Opět platí komplementarita na základě bází purinové a pyrimidinové, ve kterých znak **A** doplní znak **T** a obdobně znak **C** doplňuje znak **G**. `5´ CAGCTGGTGCACATTTGTTGG 3´` a druhý řetězec `3' CCAACAAATGTGCACCAGCTG 5´` nemohou být považovány za úspěšný nález, neboť ve směru čtení řetězců od `5´` konců jsou řetězce komplementární. K zadanému řetězci lze druhý nalezený řetězec akceptovat, pokud toto pravidlo neporuší o více než 30%.

2. Pravidlo posledních tří nukleotidů - Poslední 3 nukleotidy u 3´konců obou řetězců nesmí být komplementární vůbec.

3. Pravidlo posledních pěti nukleotidů - Posledních 5 nukleotidů od 3´konce musí obsahovat alespoň 2 pozice A či T,

4. Pravidlo Hybridizační teploty - oba nukleotidy musí mít hybridizační teplotu T v rozmezí 50-60°C. Hybridizační teplota je dána množstvím jednotlivých

bází, kdy každý výskyt báze **C**ytosinu nebo **G**uaninu přidá celkové hybridizační teplotě 4°C. Obdobným způsobem se zhodnotí i báze **A**deninu a **T**hyminu, ovšem tyto báze navyšují celkovou teplotu pouze o 2°C. Bývá též zvykem, že po spočtení celkové teploty se odečte 5°C, čímž lze mírně prodloužit oba řetězce - a tím je i lépe specifikovat, není to však nezbytně nutné.

5. Pravidlo vzdálenosti oligonukleotidů - Vzdálenost mezi oběma oligonukleotidy je v rozmezí 100 – 250 nukleotidů.

6. Pravidlo o prostoru mezi oligonukleotidy - Sekvence DNA uprostřed mezi dvěma oligonukleotidy by měla obsahovat co nejvíce A, T nukleotidů.

7. Pravidlo sobecké DNA - Při vyhledávání lze odfiltrovat tzv. sobeckou DNA (předem známé sekvence DNA bez biologicky využitelných funkcí) a dlouhé opakující se motivy.

4. Použití vybraných metod pro oblast DNA

4.1. Aplikace pravidel PCR procesů z hlediska časové náročnosti

Nejprve bych se krátce vrátil k pravidlům pro specifikaci hledaných řetězců tak, jak jsou definovány v minulé kapitole. V minulé kapitole jsou sepsány pohledem pracovníka genetické laboratoře a tak, jak je tento pracovník používá. Po krátké úvaze ovšem vyvstává otázka, zda-li je vhodné, či vůbec možné, použít tato pravidla v nezměněném tvaru i při vyhledávání pomocí informační technologie.

Při převodu prvního pravidla je nutné si uvědomit, že jeden z oligonukleotidů bude zadán laborantem a tento řetězec musí být nutně i součástí nalezeného úseku. Když bude zadán řetězec, který se ve zdrojovém kódu nevyskytuje, není tedy nutné ověřovat další podmínky, a hledání skončí. V případě nálezu však bude nutno najít odpovídající řetězec v okolí zadaného (pravidlo 5). Uvážíme-li nyní, že rozmezí pro výskyt tohoto řetězce je 2x 150 znaků, bylo by jeho vyhledávání velice náročné, pokud by bylo ponecháno jako součást hledání řetězce zadaného. Mnohem výhodnější z hlediska výsledného času se jeví ponechat prvotní průběh nepřerušen a pouze si zaznamenat pozice výskytů zadaného řetězce.

Dohledávaný řetězec navíc není přesně určený, neboť může být zatížen až 30% chybou. Již na první pohled by tedy dohledávání odpovídajícího řetězce mohlo být náročné. Časovou náročnost tohoto problému nejlépe ukážeme, když podle pravidla o hybridizační teplotě (pravidlo číslo 4) spočítáme minimální a maximální délku řetězců. Minimální ohodnocení báze je rovno dvěma a maximální přípustná teplota je 60°C, z čehož vychází maximální délka řetězce rovna 30 znakům. Obdobným způsobem vyjde minimální délka, rovna 14. Pokud bychom procházeli okolí zadaného řetězce znak po znaku, neshodu by bylo možné s jistotou detekovat v nejlepším případě nejdříve po pátém (30% ze 14 je přibližně 5) prozkoumaném znaku, pravděpodobněji však mnohem později.

Zde však lze využít pravidel o několika posledních znacích nalezených řetězců (jedná se o pravidla 2 a 3) a jelikož je původní řetězec pevně zadaný, vztahuje se toto pravidlo právě na dohledávaný řetězec. I když ani tato dvě pravidla nedefinují jednoznačný řetězec, pro který by bylo možno využít některou z metod pro nalezení přesné shody metod, lze chybu detekovat po třech, respektive pěti znacích, čímž se hledání též urychlí.

Z dosud popsaných pravidel je možné určit všechny vyhovující dvojice oligonukleotidů, pro proces PCR je však nutné vybrat ten nejvhodnější. Budeme-li předpokládat, že všechna umístění kdekoliv ve zdrojovém řetězci jsou si rovna

náročností dalšího zpracování, lze za pomocí pravidla o prostoru mezi nukleotidy (pravidlo 6) sestavit pořadí vyhovujících výskytů a výsledky jednoduše seřadit.

Poslední zbývající pravidlo je pouze teoretické, protože dostupné DNA databáze již tuto „sobeckou DNA“ neobsahují a stejně tak jsou zkráceny o dlouhé opakující se motivy. Její teoretické využití tedy připadá pouze na zcela nově objevené úseky DNA, které ještě nebyly žádným způsobem přepracovány. V této práci je tedy pro úplnost uvedeno, není však žádným způsobem řešena.

4.2. Aplikace pravidel PCR procesů z hlediska paměťové náročnosti

Až dosud je v práci popsán způsob vyhledávání pouze s ohledem na časovou náročnost. Vezmu-li v potaz, že nejkratší hledaný řetězec bude mít 14 znaků a DNA kód se skládá ze čtyř různých bází, je pravděpodobnost jeho výskytu $1:4^{14}$ což odpovídá poměru 1:268 435 456. Přestože hledaný řetězec není vybírán náhodně, čili jde předpokládat výskyt častější než v tomto poměru, jeho četnost bude přesto nízká. Zaznamenání všech teoretických výskytů řetězce bude tedy z hlediska paměťové náročnosti zcela zanedbatelné.

4.3. Postup zpracování

Celý proces vyhledávání se nyní rozdělil do čtyř kroků:

1. **Nalezení zadaného řetězce ve zdrojovém kódu a zaznamenání jeho pozice -** V tomto kroku dojde k průchodu zdrojovým řetězcem podle předpisů zvolené metody, nalezení všech případných shod ve zdrojovém řetězci se zadaným a zaznamenání pozice začátku řetězce ve zdrojovém kódu.

2. **Nalezení vhodného odpovídajícího řetězce pro každou ze zaznamenaných pozic –** V tuto chvíli již disponujeme seznamem všech nalezených shod zadaného řetězce. Pro využití v optimalizaci PCR procesu je však vyžadován ještě druhý řetězec. V tomto bodě tedy budou postupně procházeny všechny nalezené shody se snahou nalézt druhý řetězec, jež zároveň bude splňovat nutná kritéria. Shody, k nimž není možné přiřadit žádný řetězec, jsou ze seznamu shod odebrány. Cílem tohoto kroku je stanovit seznam dvojic indexů – počátku zadaného a počátku odpovídajícího řetězce.

3. **Stanovení počtu znaků podle pravidla o prostoru mezi nukleotidy –** K seznamu vyhovujících dvojic přibude nyní ještě informace o počtu znaků **A**, či **T** mezi zadaným a k němu odpovídajícím řetězcem. V tuto chvíli je seznam nalezených shod podle kroku č.1 již tříslożkový a obsahuje dva indexy do textového řetězce a číslo v rozmezí 0-250.

4. **Seřazení výsledků a jejich prezentace -** Podle třetí složky v seznamu, tedy počtu bází **A**deninu a **T**hyminu mezi oligonukleotidy se záznamy seřadí. První záznam se posléze vybere a je prezentován uživateli.

Z důvodu vyhnutí se vícenásobnému průchodu zdrojovým řetězcem probíhají kroky 2 a 3 během jednoho průchodu a do značné míry současně.

4.4. Možnosti prezentace výsledků

Nalezením shody a ověřením její platnosti vůči všem pravidlům však zpracování nekončí, výsledek, který by byl tvořen třemi číslicemi, zdánlivě řazených beze smyslu, by jistě nebyl příliš k užitku.

4.4.1. Prezentace s výsekem kódu

S využitím již zmiňovaného rozdělení úseku DNA po deseti znacích a šedesáti znacích na řádku, lze dosáhnout přehledného a jasného výsledku. Je však třeba mít na paměti, že nalezený záznam bude zřídka kdy začínat na pozici dělitelné šedesáti beze zbytku, a i přes uvedení pozice na začátku řádku je vhodné začátek textu odsadit právě o tento zbytek.

Pro zřetelné zpracování výsledků je vhodné použít font, který má neměnnou velikost znaků (tedy například Courier New), navíc je dobré zadaný a k němu odpovídající řetězec zvýraznit (zde se nabízí například podtržení a vypsání kurzívou).

Výsledek může být prezentován třeba takto:

```
124     acgcct atgcgtttac aacacggatc ggatttgtgg aaatcggcta atcgactgtg
180 ctatgtttag tacaacgagg ctgcctgtgg gaatcagatc ggatcatcca atcggctgtg
240 ctat
```

Tento druh výpisu je také použit v programové části práce.

4.4.2. Prezentace s výpisem oligonukleotidů

Výpis s výsekem kódu postrádá jednu vlastnost. Není v něm totiž možné vypisovat druhý nalezený řetězec tak, jak se vyskytuje na druhém vlákně DNA, tedy v negaci ke kódu zadanému. Z tohoto důvodu lze využít ještě výpis, kde budou přepsány oba řetězce, jeden zadaný a nezměněný, druhý převedený na tvar z druhého vlákna DNA. Při tomto způsobu však není dost dobře možné vypisovat i obsah prostoru mezi oligonukleotidy, a laborant, který bude s výsledkem pracovat, je tedy zcela odkázán na údaje jemu předložené, bez možnosti ověřit si jejich správnost.

Výsledek může být prezentován jako:

```
Sekvence A (zadáno):    5‘ acgcctatgcgtttacaaca 3‘, pozice 12 451
Sekvence B:             3‘ tgttgtaaacgcataggcgt 5‘, pozice 12 322
```

5. Vybrané algoritmy

Seznam algoritmů popsaných v této kapitole není a ani nebyl zamýšlen jako vyčerpávající seznam všech algoritmů pro vyhledávání přesné shody znaků. Jedná se pouze o popis vybraných algoritmů, které jsou nějakým způsobem relevantní k problematice DNA a jeho zpracování.

Definice používaných termínů

Pro lepší pochopení jednotlivých algoritmů je nutné definovat si některé využívané pojmy. Aby vůbec bylo možné vyhledávat, je nutné mít nějaký **zdrojový** řetězec (anglicky též Source String) o délce *n* znaků, tedy text, ve kterém se bude vyhledávat. Nyní víme, kde budeme vyhledávat, ještě však musíme mít něco, co chceme vyhledat. Takový řetězec (slovo, větu, či jiný krátký text) označíme jako **vzorový řetězec (**anglicky Pattern String) o délce *m* znaků. Nyní již lze zahájit vyhledávání, ovšem z důvodu širokého využití se bude hodit definování ještě následujícího pojmu. Půjde o pojem **vyhledávací okénko**. Vyhledávací okénko představuje takovou část zdrojového řetězce, na které se v jednom kroku provádí porovnání se zadaným řetězcem. Délka vyhledávacího okénka je tedy rozsah znaků, ke kterým v rámci jednoho kroku, neboli též v jednom pokusu, může algoritmus přistupovat a pracovat s nimi. Z výše uvedeného vyplývá, že vyhledávací okénko bude vždy minimálně jeden znak veliké, ovšem zpravidla bývá jeho velikost stejná jako délka zadaného řetězce.

Sestavení testovacích dat

Pro účely testování byl použit řetězec o délce 91689 znaků. Tento řetězec reprezentuje skutečný DNA záznam, konkrétně jde o mutaci zdravého DNA kódu po žloutence [8],[9]. Testy probíhali na počítači s následující konfigurací: Procesor AMD Sempron 3100+, 1GB DDR2 RAM, GeForce 7600GS (256MB), MS Windows XP SP3.

Hledaný výraz se pak mění tak, aby bylo možné názorně předvést rozdíly mezi jednotlivými algoritmy. První řetězec popisuje předpokládaný běžný případ, neboť je tvořen částí DNA kódu, konkrétně prvních 18-28 znaků ze zdrojového řetězce - čili „cctaaatagttcatcacaTGAGAAGCTA“. Znaky psané velkými písmeny představují znaky, které se v závislosti na délce zkoušeného řetězce nemusí vyskytovat.

Další dva vzorky jsou pak uměle vytvořené (avšak jejich reálná existence je stále pravděpodobná), poslední vzorek pak obsahuje pouze znaky, které se v DNA nemohou vyskytovat. Cílem testování i tohoto řetězce je vytvoření pomyslné hranice časové náročnosti algoritmů a zároveň změření závislosti časové náročnosti na délce hledaného řetězce tak, aby nemělo vliv složení (myšleno použité znaky) hledaného řetězce.

Všechny naměřené výsledky jsou uváděny v nanosekundách a v grafech jsou pro lepší přehlednost zaokrouhleny na milisekundy.

5.1. Brute Force

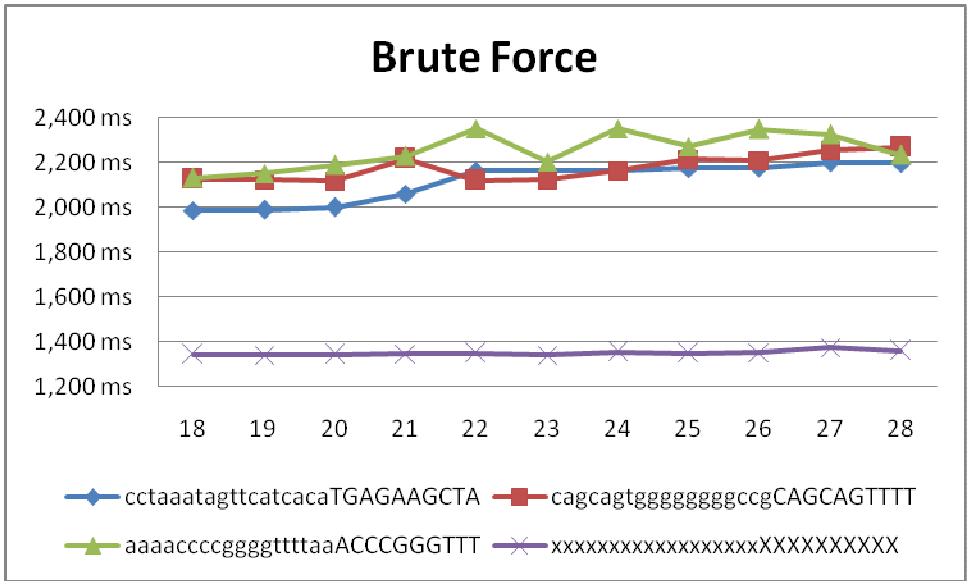
5.1.1. Vlastnosti algoritmu

Zpracování textu metodou Brute Force [9] je patrně nejstarší a nejzákladnější přístup k automatickému zpracování textu. Je založen na principu „každý s každým“, čili porovnává znaky ve zdrojovém i vzorovém řetězci vzájemně tak dlouho, než narazí na neshodu znaků, nebo na konec zdrojového řetězce. Tento postup by sám o sobě nebyl ještě úplně nevýhodný, neboť by v zásadě znamenal jen jeden průchod zdrojovým řetězem a tedy složitost ***O(n)***. Bohužel metoda Brute Force používá vyhledávací okénko velikosti právě jedna a tak v případě neshody i v případě nalezení zadaného řetězce může provést posun jen o jeden jediný znak. V nejhorším případě tedy bude nucena provést *n* x *m* znakových porovnání, čímž se její časové nároky dostávají do závratných výšin.

5.1.2. Experimentální vyhodnocení

Z výše popsaného lze odhadnout chování metody při použití v praxi. Vzhledem k posunu vždy o jedno políčko lze očekávat téměř konstantní dobu vyhledávání pro všechny hledané řetězce s výjimkou posledního, kde bude složitost opravdu zmíněných ***O(n)***. Pro jednotlivé zkoušené řetězce se časy průběhu samozřejmě mohou mírně lišit v závislosti na svém složení, nicméně ze statistického hlediska by tyto vzájemné rozdíly mohly být velmi malé, limitně takřka nulové.

I při minimální délce vzorového řetězce, kterou je v testovaných případech 18 znaků, lze předpokládat, že neshoda nastane dříve a algoritmus tedy provede posun okénka dříve nežli na osmnáctém znaku. Závislost na délce vzorového řetězce bude tedy opět malá.



Graf 5.1.1 Časový průběh algoritmu Brute Force

Z grafu Graf 5.1.1 jasně vyplývá, že samotný průběh zdrojovým řetězcem má konstantní dobu zpracování. Značí to křivka pro čtvrtý vzorový řetězec, který obsahuje jen takové znaky, které se v DNA nevyskytují a neshoda je tudíž detekována okamžitě.

Na křivkách ostatních řetězců se potvrdil předpoklad nezávislosti algoritmu na složení a délce vzorového řetězce v DNA aplikacích. Všechny měřené řetězce dosáhly téměř totožných časových nároků a rozdíl řetězců délky 28 a řetězců délky 18 znaků je zanedbatelný.

Algoritmus Brute Force je tedy nezávislý na vzorovém, či zdrojovém řetězci, ovšem jeho časová náročnost je poměrně vysoká ve všech případech.

Teoretické využití by algoritmus mohl mít v případech, kdy se výskyt hledaného řetězce očekává blízko začátku databáze a po první nalezené shodě již není nutné další prohledávání. V těchto případech by tudíž mnohdy složité a časově poměrně náročné předzpracování textu, které využívají ostatní algoritmy, mohly zabrat více času, než prosté projití textu. Z tohoto důvodu je algoritmus použit například pro dohledávání odpovídajících řetězců při zpracování v kroku 2 kapitoly 4.3.

5.1.3. Shrnutí

- Časová náročnost je v nejhorším případě **O**(*mn*).
- Porovnává všechny znaky s posunem o jeden znak v každém kroku.
- Nevyužívá žádnou dodatečnou paměť.

5.2. Karp-Rabinův Algoritmus

5.2.1. Vlastnosti algoritmu

Náročnost porovnávání znaků způsobem, jak to dělá Brute Force je zjevně neefektivní. Porovnávání znaků samo o sobě je poměrně náročné, neboť je nutné tento znak nejprve dekódovat podle použité znakové sady, určit zda se jedná o písmeno velké, či malé, apod. Proto přišli Richard M. Karp a Michael O. Rabin [10] s algoritmem, který není založen na porovnávání znaků, nýbrž porovnává jejich číselné reprezentace.

Pro popis Karp-Rabinova algoritmu je nutné zavést pojem (re)hash funkce. **Hash funkcí** se rozumí posloupnost příkazů, s jejichž pomocí lze jednoznačně převést zadaná data do (relativně) malého čísla. Podstatné je, že pro dvoje různá vstupní data musí být výsledné číslo různé a analogicky pro stejná data se výsledná čísla musí shodovat. **Funkce Rehash** bude podobným způsobem převádět data na čísla, ovšem s tím rozdílem, že zpracuje jen změnu oproti současné hodnotě.

V přípravné fázi pouze spočítá Hash funkce hodnotu vzorového řetězce. Je tedy evidentní, že časová náročnost přípravné fáze je **O**(m).

V průběhu vyhledávání se zjištěná hodnota porovnává s hodnotou vypočtenou pomocí funkce Rehash. V Karp-Rabinově algoritmu pracuje funkce Rehash na základě dvou vlastností:

a) Rehash se provádí při každém posunu okénka, řídící změnou je znak na této nové pozici.

b) Délka zadaného řetězce je pevně daná a neměnná.

Rehash funkce tedy pokaždé nejprve umocní současnou hodnotu základem číselné soustavy (Kupříkladu při použití klasické desítkové soustavy je umocnění totéž jako vynásobení číslem 10), přičte hodnotu nového znaku a poté „zapomene“ x-tou mocninu vypočteného čísla, kde X je délka vzorového řetězce. Pro vlastní naprogramování jsem použil kruhový spojový seznam, jelikož prosté zapisování číslice se jeví efektivnější, nežli výpočty s vysokými mocninami a použití (vysokých hodnot) Integer čísel.

A=1 C=2 G=3 T=4

A	C	T	G	A	A	G	A
1	2	4	3	1	1	3	1
1	2	4	3				
	2	4	3	1			
		4	3	1	1		
			3	1	1	3	
				1	1	3	1

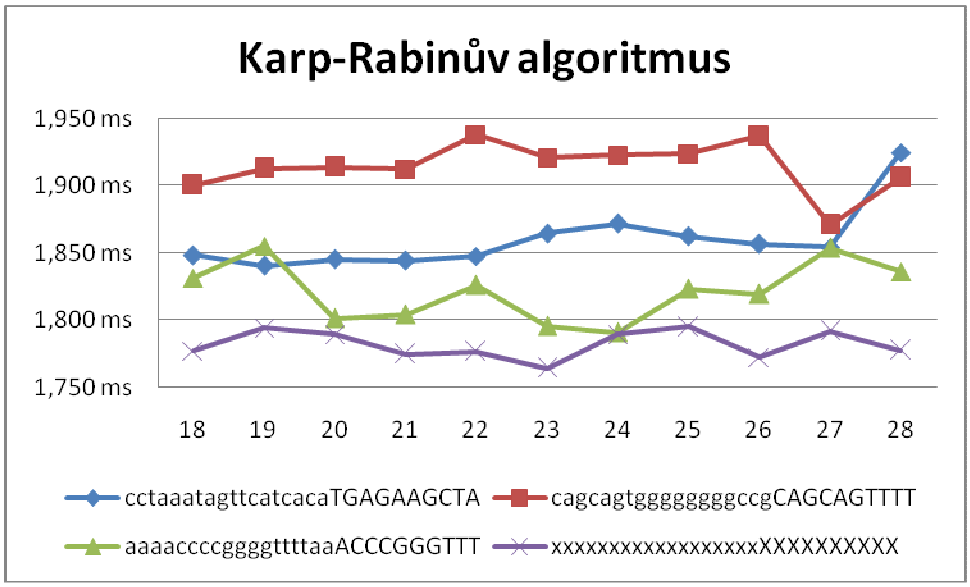
Tabulka 5.2.1 Postup Karp-Rabinova algoritmu

Získané hodnoty znaků se poté porovnávají mezi sebou. V případě shody je však stále nutné ověřit správnost shody porovnáním znaku po znaku, ovšem jen v omezené míře, jelikož stačí porovnat pouze *m* posledně zpracovaných znaků.

Výhodou Karp-Rabinova algoritmu je možnost vytvořit Hash funkci na míru cílovému použití, kdy je možné zavést například míru podobnosti znaků (znaky „a“ a „g“ lze například pro účely cílové aplikace možné prohlásit za totožné), nebo urychlit vyhledávání tím, že se stará hodnota bude rehashovat až po každém n-tém znaku. Nicméně tímto postupem většinou utrpí přesnost výsledku.

5.2.2. Experimentální vyhodnocení

Jelikož Brute Force i Karp-Rabinův algoritmus posouvají vyhledávací okénko vždy jen o jeden znak, lze u obou metod očekávat podobný průběh. Relativní nezávislost řetězců na svém obsahu je zde dána právě funkcí Hash, závislost na délce vzorového řetězce očekávejme ze stejného důvodu taktéž nepatrnou.



Graf 5.2.1 Časový průběh Karp-Rabinova algoritmu

I v tomto případě se původní domněnky potvrdily, ovšem rozdíly mezi řetězcem neplatných znaků a ostatními je výrazně nižší. To je patrně dáno tím, že porovnávání znaků se provádí jen v případě shody hashových hodnot a počet jejich shod je ve zdrojovém DNA kódu nízký. Tato skutečnost se projeví i ve sníženém celkovém času jednotlivých vyhledávání.

Ani Karp-Rabinův algoritmus tedy není závislý na tvaru a délce zdrojového nebo vzorového textu.

V DNA aplikacích je vhodné použít Karp-Rabinův algoritmus ve chvílích, kdy není nutné vyhledávat zcela přesnou shodu, nebo je potřeba jen přibližná pozice shody, jelikož v takovém případě se lze spokojit s pouhým porovnáním ve formě číslic a není

nutné provádět porovnání jednotlivých písmenkových znaků. Alternativně je navíc možné využít Karp-Rabinův algoritmus pro aplikace, které se spokojí s rozlišením purinová-pyrimidinová náze

5.2.3. Shrnutí

- Časová náročnost je **O**(*mn*) ve vyhledávací fázi a **O**(*m*) v přípravné fázi.
- Předpokládaný čas běhu je ve většině případů **O**(*n* + *m*).
- Algoritmus lze snadno přizpůsobit konkrétním požadavkům cílové aplikace.
- Porovnává číselné hodnoty namísto porovnání znak-znak

5.3. Knuth-Morris-Prattův algoritmus

5.3.1. Vlastnosti algoritmu

Oba doposud zpracované algoritmy posouvaly vyhledávací okénko vždy jen o jedno políčko. Je však možné posunovat okénko o více než jeden znak? Donald Knuth a Vaught R.Pratt a nezávisle na nich i J.H.Morris v roce 1977 dokázali, že ano.

Při publikování jejich algoritmu [9],[11], který, ač vymyšlen nezávisle, bývá označován jmény všech tří vědců nebo zkratkou KMP, zavádí pojmy **prefix** a **suffix**. V obou případech jde o unikátní část řetězce, liší se pouze směrem, ze kterého jsou zjišťovány – prefix zleva doprava, tak jak jsme zvyklí číst obyčejný text, a suffix naopak od konce směrem k začátku. Slovo „*pokus*“ bude mít tedy prefixy „*p*“, „*po*“, „*pok*“, „*poku*“ a prázdný řetězec, zatímco suffixy budou „*s*“, „*us*“, „*kus*“, „*okus*“ a samozřejmě vždy také prázdný řetězec.

Se znalostí všech prefixů a suffixů vzorového řetězce můžeme jednoduše stanovit takovou část řetězce, že znaky, ze kterých se skládá, jsou zároveň prefixem i suffixem. Taková část se nazývá **hranice** (anglicky border) řetězce. Například hranice řetězce „*acabac*“ bude obsahovat „*ac*“ a prázdný řetězec.

V přípravné fázi KMP algoritmus spočítá délku nejdelší hranice postupně pro celý vzorový řetězec tak, že vždy odebere poslední znak a nově vzniklý řetězec považuje za celý řetězec. Výsledkem zmíněného postupu je tabulka délek hranic řetězce snížená o jedna (čímž se zajistí posun až za zjištěnou hranici) pro každou pozici v řetězci.

Knuth později algoritmus ještě vylepšil dodatečnou podmínkou, že znak ihned za nalezenou hranicí se musí lišit, pokud se chceme vyhnout další okamžité neshodě. Zatímco bez použití této podmínky by se v řetězci „*gagagatct*“ při vzorovém řetězci „*gatc*“ po případné neshodě na třetím znaku okénko posouvalo o dvě a ihned by došlo k další neshodě, protože následující porovnání je opět „T-G“, nyní víme, že můžeme využít posunu až na druhý výskyt hranice tedy posun o čtyři.

Ve vyhledávací části navíc algoritmus na rozdíl od Karp-Rabinova, či Brute Force registruje délku dosud shodné části. Ve chvíli, kdy nastane neshoda, stačí od pozice vzniklého konfliktu odečíst délku hranice pro tuto pozici a vyhledávací okénko posunout o výslednou vzdálenost.

Tabulka KMP

0	1	2	3	4	5
A	**G**	**C**	**A**	**G**	**T**
-1	0	0	-1	0	2

Tabulka 5.3.1 Posuny KMP algoritmu

0	1	2	3	4	5	6	7	8
A	G	C	A	G	C	A	G	T
A	G	C	A	G	T			
			A	G	C	A	G	T

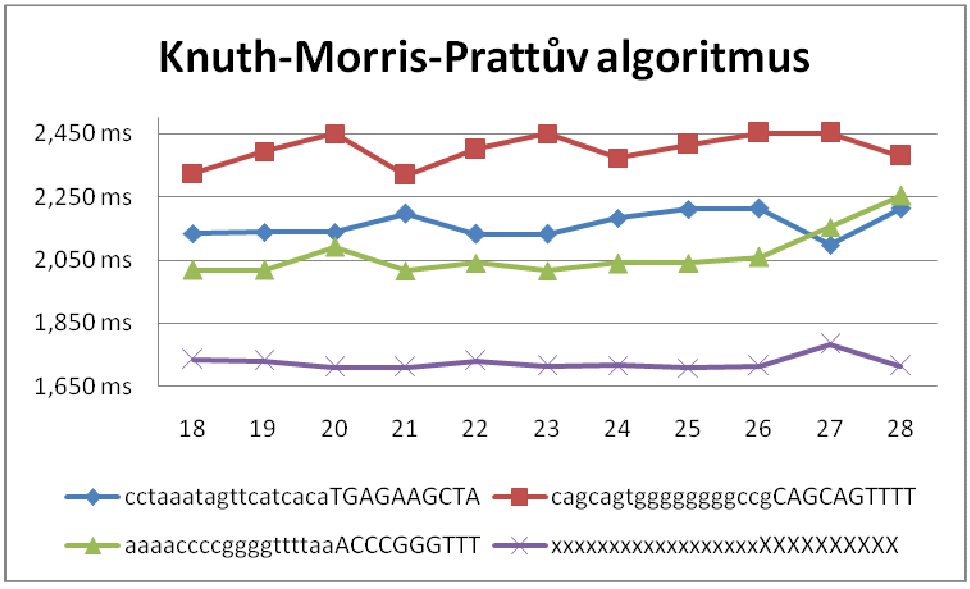
Tabulka 5.3.2 Postup KMP algoritmu

Tabulka 5.3.2 Postup KMP algoritmu znázorňuje postup metody pro řetězec „*agcagt*“. Při prvním pokusu nastane neshoda až na pozici 5. Z tabulky KMP zjistíme, že hranice pro tuto pozici je rovna číslu 2 a okénko tedy lze posunout o 5-2 = 3 znaky. Víme, že prvních pět znaků je „*agcag*“, znak na pozici 5 (což je vlastně šestý znak) však není „*t*“. Jelikož opakovat se mohou jen dva znaky a sice „*ag*“, první tři znaky není nutné porovnávat znovu.

5.3.2. Experimentální vyhodnocení

Vzhledem k posunovací tabulce algoritmu bude patrně dobu běhu ovlivňovat složení vzorového řetězce. Řetězec, který bude obsahovat opakujících se části, bude zřejmě vyhledán rychleji, poněvadž při neshodě můžeme vyhledávacího okénko posunout nejvíce.

Na druhou stranu není důvod domnívat se, že by délka zadaného řetězce výrazněji ovlivnila časové nároky KMP algoritmu.



Graf 5.3.1 Časový průběh Knuth-Morris-Prattova algoritmu

Nezávislost délky řetězce je opět jasně patrná z grafu, ve všech zkoušených případech je doba běhu v závislosti na počtu znaků hledaného řetězce neměnná.

Závislost na složení zadaného řetězce se oproti očekávání příliš neprojevila. To si lze vysvětlit tím, že v DNA nastane neshoda často v několika málo prvních znacích při kterých nejsou hranice nijak veliké. Díky tomu posuny vyhledávacího okénka nejsou

oproti předchozím algoritmům příliš rozdílné, což se podepisuje na podobných křivkách průběhu.

Budeme-li potřebovat nalézt řetězec takový, který bude tvořen například stále dokola opakujícími se znaky (tedy kupříkladu „*gagagagaga*“) mohla by tato metoda dosahovat již zajímavých výsledků.

5.3.3. Shrnutí

- Časová náročnost je **O**(*mn*) ve vyhledávací části a **O**(*m*) v přípravné fázi.
- Předpokládaná časová náročnost je ve vyhledávací části **O**(*m* + *n*).
- Registruje některé opakující se části ve vzorovém řetězci a využívá jich k řízení posunu vyhledávacího okénka

5.4. Boyer-Mooreův algoritmus

5.4.1. Vlastnosti algoritmu

Dá se říci, že Boyer-Mooreův algoritmus [11] je v dnešní době nejrozšířeněji používaný algoritmus. V různých podobách je používán ve většině textových aplikacích v příkazech „*najít*“, případně „*najít & nahradit*“ (například MS Word, Notepad, apod.) a je od něj odvozeno i velké množství dalších algoritmů. Algoritmus vyvinutý Bobem Boyerem a J. Strother Moorem [12] se zaměřuje především na rozsáhlé abecedy, tedy zdrojové řetězce, které se skládají z mnoha různých znaků. Použití v DNA je tedy značně omezené, nicméně, tento algoritmus dal vzniknout mnoha dalším algoritmům a považuji za vhodné rozebrat v této kapitole jeho principy.

Zcela zásadní změnou oproti předchozím algoritmům je porovnávání znaků ve vyhledávacím okénku zprava doleva, zatímco vyhledávací okénko se posouvá stále směrem zleva doprava.

Uvážíme-li, jaké mohou nastat případy při jednom pokusu, zjistíme, že může nastat shoda, čili nalezení vzorového řetězce, nebo dva různé druhy chyby. První druh je takový, kdy narazíme ve zdrojovém řetězci na znak, který se ve vzorovém vůbec nevyskytuje. V tomto případě lze snadno odvodit, že při jakémkoliv vyhledávacím okénku, kdy tento znak bude obsažen, nemůže shoda nikdy nastat. Protože Boyer-Mooreův algoritmus porovnává znaky odzadu, můžeme ušetřit porovnání až *m-1* znaků (pokud takový znak detekujeme hned při prvním porovnání), stačí vyhledávací okénko okamžitě posunout za tento neplatný znak. Tomuto typu chyby se říká **chyba špatného znaku** (anglicky *Bad-Character Shift*).

Druhou možností je neshoda, kdy však vzorový řetězec obsahuje znak, který neshodu způsobil. Zde opět zužitkujeme výhody porovnávání od konce. Předpokládejme, že k neshodě došlo až po projití několika znaků, tedy částečné shodě i znaků (označme tuto shodu jako suffix u), přičemž pokud by $i = 1$ (čili neshoda na prvním zkoumaném znaku) můžeme posunout okénko jen o jeden znak doprava. Pokud tedy $i > 1$ víme, že minimálně i znaků je obsaženo v obou řetězcích. Stačí tedy zarovnat vzorový řetězec na nejpravější výskyt suffixu u, abychom však nezarovnávali stále na jeden a tentýž sufix, musí novému zarovnání předcházet jiný znak, nežli ten, na kterém nastala neshoda. Posun tedy proběhne na nejpravější **správný suffix** (anglicky *Good-Suffix Shift*). Ačkoliv na první pohled to nevypadá, toto je stejná situace, kterou zpracovává již Knuth-Morris-Prattův algoritmus s jedinou výjimkou, a sice zde jde o posun v opačném směru [7].

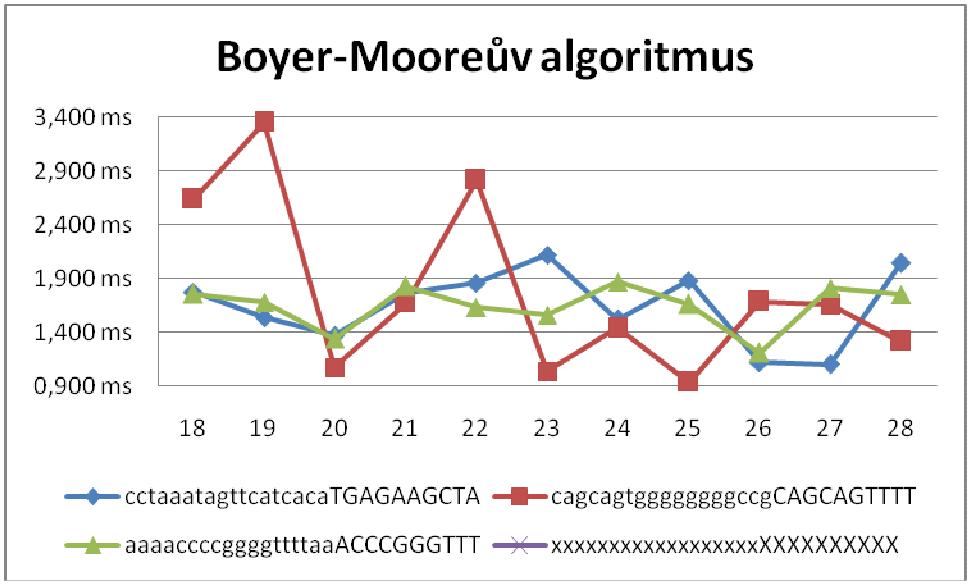
Pro uchování informací o obou druzích chyby je nutné v přípravné fázi vytvořit tabulku o rozsahu rovném délce vzorového řetězce, neboli m, a druhou o rozsahu rovném rozsahu abecedy, ten se značí o. Paměťové nároky jsou tedy evidentní: $\mathbf{O}(m + o)$.

5.4.2. Experimentální vyhodnocení

Boyer-Mooreův algoritmus byl designován zejména s ohledem na rozsáhlé abecedy a tak nízký počet znaků, ze kterých se skládají DNA kódy omezuje jeho efektivitu.

Zcela jednoznačně lze očekávat závislost zejména na složení vzorového řetězce, neboť pokud zvolíme takový, který neobsahuje některé znaky zdrojového řetězce, urychlení bude značné.

Závislost na délce řetězce bude ve většině případů nabízet jen delší posun v závislosti na *chybě špatného znaku*. Pokud se tato závislost projeví, bude její vliv malý.



Graf 5.4.1 Časový průběh Boyer-Mooreova algoritmu

Graf 5.4.1 a výkyvy jeho křivek naznačují závislost na řetězcích, nad kterými algoritmus probíhá. Pokud jednotlivými křivkami proložíme regresivní přímku, míra závislosti na délce vzorového řetězce vyjde téměř nulová. Výkyvy křivek tedy způsobuje složení řetězce.

5.4.3. Shrnutí

- V přípravné fázi je časová i paměťová složitost $\mathbf{O}(m + o)$, kde o je rozsah abecedy znaků.
- Časová náročnost je $\mathbf{O}(mn)$ ve vyhledávací části v nejhorším případě, pro rozsáhlé abecedy se ale blíží k $\mathbf{O}(n / m)$.
- Porovnává znaky zprava doleva.
- Využívá znaky neobsažené ve vzorovém řetězci k rychlým posunům.

5.5. Zhu-Takaokův algoritmus

5.5.1. Vlastnosti algoritmu

Zhu-Takaokův algoritmus [13],[14] je jedním z mnoha algoritmů založených na principech Boyer-Moorova postupu. Na rozdíl od něj byl svými stvořiteli Rui-Feng Zhuem a T.Takaokou vyvíjen s ohledem na malé abecedy.

Zhu-Takaokův algoritmus přebírá beze změn výhody porovnávání od konce a část zvanou *správný suffix*. Původní algoritmus v malých abecedách doplácel na to, že zdrojový řetězec je složen z mála různých znaků, a tedy metoda pro *chybu špatného znaku* byla využita jen zřídka. V ideálním případě bychom tedy potřebovali nějaký další znak, který by zvýšil pravděpodobnost výskytu chyby špatného znaku. Autoři tedy namísto porovnání jednoho znaku porovnávají znaky dva a takto vzniklou dvojici znaků dále pro potřeby algoritmu považují za nový znak. Jednoduchým matematickým výpočtem nyní v konkrétním případě DNA z pěti používaných znaků (**A**, **C**, **G**, **T** a **N**) získáme množinu $5^2 = 25$ různých znaků. Tabulka 5.5.1 a Tabulka 5.5.2 zachycuje zpracování ukázkového řetězce „*gcagagag*“.

Tabulka Chyby špatného znaku

	A	**C**	**G**	**T**
A	8	8	2	8
C	5	8	7	8
G	1	6	7	8
T	8	8	7	8

Tabulka 5.5.1 Chyba špatného znaku

Tabulka Chyby správného suffixu

i	0	1	2	3	4	5	6	7
	G	**C**	**A**	**G**	**A**	**G**	**A**	**G**
	7	7	7	2	7	4	7	1

Tabulka 5.5.2 Chyba správného suffixu

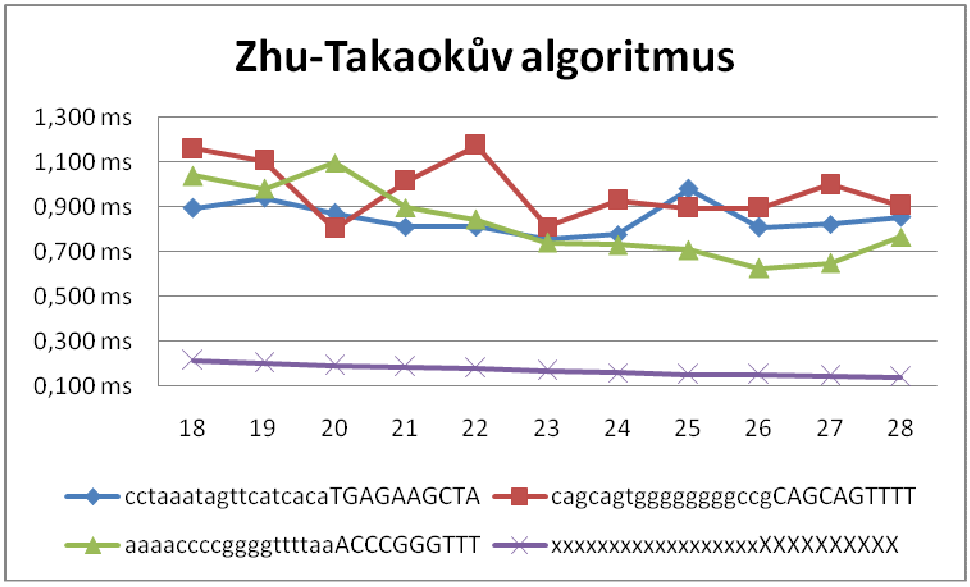
Vyhledávat v tabulce podle takto vzniklých pseudoznaků přímo nelze. Efektivním řešením je namísto jednorozměrné tabulky vytvořit dvourozměrné pole, kde indexy budou v obou směrech postupně všechny znaky abecedy. Tím je zajištěno pokrytí všech možných pseudoznaků i rychlé vyhledávání velikosti možného posunu. Vedlejším efektem je však vyšší paměťová náročnost, jelikož již neroste lineárně podle rozsahu abecedy, ale kvadraticky. To je však přijatelné, neboť tato metoda byla navrhována pro malé abecedy a očekávané paměťové nároky jsou stále relativně nízké.

5.5.2. Experimentální vyhodnocení

Jelikož je Zhu-Takaokův algoritmus přímo odvozen od Boyer-Mooreova algoritmu není důvod očekávat jiné závislosti. Připomenu, že u Boyer-Moorova algoritmu byla

očekávána a následně experimentálně potvrzena závislost na složení řetězců, závislost na délce řetězců se spolehlivě neprojevila.

Při experimentech nad DNA kódem je však rozumné očekávat výrazně lepší časy zpracování, jelikož Zhu-Takaokův algoritmus byl navržen speciálně pro podobné případy.



Graf 5.5.1 Časový průběh Zhu-Takaokova algoritmu

Zdá se, že i zde se potvrdila závislost na složení řetězce, jelikož řetězce v grafu Graf 5.5.1 mají podobné výchylky jako předcházející Boyer-Mooreův algoritmus. To není nic překvapivého, jelikož výkonný základ obou metod je taktéž velmi podobný.

Dobře viditelná je i vyšší časová výkonnost nejen oproti Boyer-Moorovi, ale v podstatě proti všem dosud zmíněným algoritmům. Zde se opět nejedná o žádné veliké překvapení, vzhledem k principům s jakými byla Zhu-Takaokova metoda navrhována.

To ale není vše, co z grafu Graf 5.5.1 vyplývá. V tomto případě se totiž projevila i menší závislost na délce vzorového řetězce. Zřejmě čím delší řetězec, tím kratší doba zpracování. Nárůstem používané abecedy na 25 znaků se zvýšila šance na velmi výhodný posun podle *chyby druhu špatný* znak. Čím delší je řetězec, tím delší posun lze takto vykonat, neboť tímto druhem posunu vyhledávacího okénka se posouváme vždy těsně za znaky, které chybu vyvolaly.

5.5.3. Shrnutí

- Přípravná fáze má $\mathbf{O}(m + o^2)$ časovou i paměťovou náročnost.
- Vyhledávací fáze má $\mathbf{O}(mn)$ časovou náročnost v nejhorším případě.
- Zvyšuje výskyt *chyby špatného znaku* a tudíž se při reálném použití blíží časové náročnosti $\mathbf{O}(n / m)$ i pro malé abecedy.

5.6. Quick Search algoritmus

5.6.1. Vlastnosti algoritmu

Také Quick search algoritmus [9],[15], někdy též Sundayův algoritmus [16], vychází z myšlenkového pochodu Boyer-Moorova algoritmu. Na rozdíl od předešlých metod však nepřidává žádná nová pravidla ke stávajícím a naopak se snaží jej zjednodušit.

Oproti Boyer-Moorově algoritmu využívá pouze postup pro *chybu špatného znaku*, který však upravuje. Vezmeme-li v úvahu, že při výskytu jakékoliv chyby nastane posun minimálně o jedno políčko a nanejvýš o délku vzorového řetězce, musí mít znak přímo za vyhledávacím políčkem přímý vliv na vyhledávání v dalším pokusu. Pokud jednotlivé znaky vzorového řetězce na sobě navzájem nejsou nijak závislé, a tedy pravděpodobnost výskytu znaku, který se nevyskytuje ve vzorovém řetězci, je stejná uvnitř okénka jako za ním, pak bude mít tento postup oproti Boyer-Mooreovu postupu mírné zrychlení, jelikož v takovém případě je možné posunout vyhledávací okénko nejen o délku vzorového řetězce, ale i právě o tento jeden znak za ním. Pokud je za vyhledávacím okénkem znak, který se ve vzorovém řetězci vyskytuje, dojde k posunu na nejpravější výskyt tohoto znaku, přičemž tento znak bude zarovnán opět se znakem až za vyhledávacím okénkem a posun bude tedy opět minimálně o jeden znak větší, než v případě Boyer-Mooreova algoritmu.

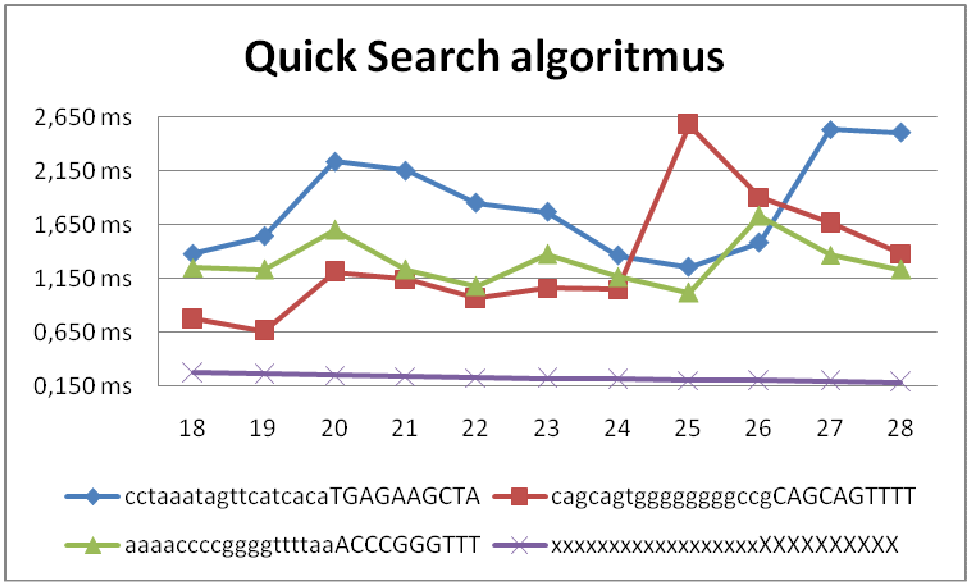
Tato změna s sebou přináší i další využitelný důsledek – není již nutné provádět porovnání směrem zprava doleva, ale naopak lze porovnávat ve zcela libovolném pořadí. Quick Search algoritmus je tak snadno upravitelný pro konkrétní řetězce, u kterých mají znaky na určitých pozicích vyšší váhu, či je na těchto pozicích větší pravděpodobnost výskytu *špatného znaku*.

Algoritmus potřebuje v paměti jen místo pro tabulku rozsahu o prvků, do které se zaznamenají nejpravější výskyty znaků použité abecedy.

5.6.2. Experimentální vyhodnocení

Stejně jako u všech ostatních algoritmů z rodiny Boyer-Moorových algoritmů je důvodné předpokládat znatelnou závislost na tvaru vzorového a zdrojového řetězce. Závislost na délce vzorového řetězce by se měla projevit jen minimálně, případně vůbec.

Oproti Boyer-Moorovi lze odhadnout mírně snížené časové nároky, přestože výrazně nižších časů by algoritmus dosahoval až při rozsáhlých abecedách.



Graf 5.6.1 Časový průběh algoritmu Quick Search

Z křivek grafu Graf 5.6.1 je odečitatelná závislost na složení řetězců. Zejména patrná je modrá (znak kolečka) a červená křivka (znak čtverečku), pro které byly řetězce složeny tak, aby při délce 26, respektive 24 znaků obsahovaly již jen minimální počet některých znaků. Rád bych také poukázal na mírně snížené časové nároky vzhledem k Boyer-Moorově algoritmu, což bylo ale také očekáváno.

Z grafu lze vyčíst mírnou závislost na délce řetězce, zejména kratší řetězce dosahovaly lepších výsledků. Minimální délka zkoušených řetězců (tj. 18 znaků) je však stále poměrně vysoká pro plnou demonstraci této závislosti.

Snadná modifikovatelnost algoritmu naznačuje použití zejména v případech, kdy jednotlivé znaky vzorového řetězce mají různou váhu. Díky tomu je vhodné, nebo nezbytné, provádět porovnání v určitém předem daném pořadí.

5.6.3. Shrnutí

- Přípravná fáze má $\mathbf{O}(m + o)$ časovou náročnost.
- Vyhledávací fáze má $\mathbf{O}(mn)$ časovou náročnost v nejhorším případě.
- Paměťové nároky jsou $\mathbf{O}(o)$
- V reálném použití se zejména pro rozsáhlejší abecedy blíží k časové náročnosti $\mathbf{O}(n / m)$, je-li vzorový řetězec krátký.

5.7. Maximal Shift algoritmus

5.7.1. Vlastnosti algoritmu

Již v případě Quick Search algoritmu bylo naznačeno, že lze provádět porovnávání znaků v libovolném pořadí a stále využívat nejpřínosnější části Boyer-Mooreova algoritmu, tedy *chyby špatného znaku*. Umožnil to přístup, kdy namísto znaku, který způsobil neshodu, se využívá znak bezprostředně za vyhledávacím okénkem a tím se vyloučí případná redundance v porovnávání stejného znaku vícekrát.

Maximal Shift [9] této vlastnosti také využívá, ovšem jde dál. Již při zadání vzorového řetězce je možné jednoznačně určit délku posunu, kterou je nutno provést v případě, že znak na této pozici se neshoduje s tím ve zdrojovém řetězci. Ve chvíli kdy vzorový řetězec je tvořen jen opakujícími se vzory, je tento posun malý, ovšem v opačném případě lze poskočit hned o několik znaků, aniž by utrpěla spolehlivost vyhledávání. Jde tedy o podobný postup jako při sestavování tabulky pro chybu *správného suffixu*.

Například při použití vzorového řetězce „*catctagaga“* jehož celková délka je 10 znaků sestaví algoritmus tabulku následovně:

Tabulka posunu Maximal Shift

0	1	2	3	4	5	6	7	8	9
c	**a**	**t**	**c**	**t**	**a**	**g**	**a**	**g**	**a**
1	2	3	3	2	4	6	2	2	2

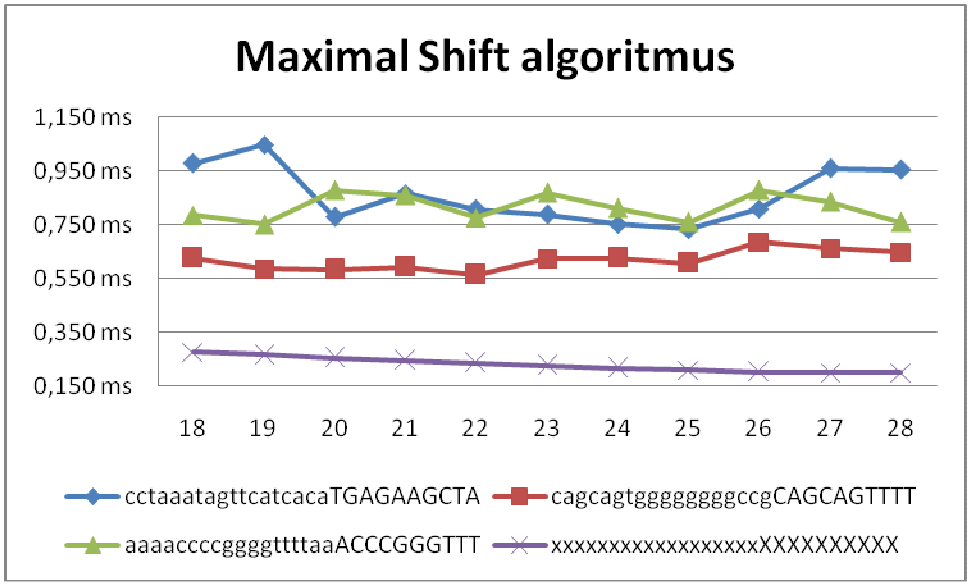
Tabulka 5.7.1 Výpočty Maximal Shift posunu

Znak „*G*“ na pozici 6 bude mít největší posun, jelikož žádný jiný znak „*G*“ se ve vzorovém řetězci před ním již nevyskytuje. Oproti tomu stejný znak na pozici 8 může posunovat jen o dvě políčka, jelikož musí provést zarovnání na další výskyt stejného znaku. Dle tabulky je vidět, že v tomto ukázkovém příkladě by pořadí porovnávání znaků bylo toto: 6, 5, 3, 2, 9, 8, 7, 4, 1, 0.

Algoritmus opět nebude mít žádné extrémní požadavky na paměť, neboť navíc oproti předchozímu algoritmu vytváří jen tabulku s pořadím prvků pro porovnání.

5.7.2. Experimentální vyhodnocení

Principem Maximal Shift postupu je seřazení znaků hledaného řetězce sestupně podle toho, které znaky mohou přinést největší posun. Je tedy zcela evidentní, že časová náročnost algoritmu bude závislá na tvaru řetězce. Dále podobně jako v případě Quick Search algoritmu, ze kterého vychází, nebude docházet k výraznému ovlivnění v závislosti na délce vzorového řetězce.



Graf 5.7.1 Časový průběh algoritmu Maximal Shift

Na grafu Graf 5.7.1 můžeme pozorovat výraznou časovou úsporu oproti algoritmu Quick Search a vlastně i téměř všem ostatním. Závislost na složení vzorového řetězce není sice zcela evidentní, nicméně zde bude na vině skutečnost že všechny testované řetězce měli pořadí sestaveno tak, že se nejdříve porovnávaly znaky mimo upravovanou část, případně ve zdrojovém řetězci na této pozici často docházelo ke shodě.

Z hlediska nízkých časových náročností je algoritmus Maximal Shift dalším kandidátem na použití při vyhledávání v běžné DNA.

5.7.3. Shrnutí

- Přípravná fáze s $\mathbf{O}(m^2+o)$ časovou náročností
- $\mathbf{O}(m+o)$ paměťové požadavky
- Vyhledávací fáze má $\mathbf{O}(mn)$ časovou náročnost v nejhorším případě
- V praxi se ovšem časová náročnost blíží $\mathbf{O}(n/m)$

6. Závěr

Jak již bylo zmíněno v úvodu, tato práce si kladla za cíl zmapovat problematiku zpracování informace v genetickém kódu a jeho propojení s možnostmi výpočetní technologie. Jednalo se především o analýzu různých způsobů prohledávání textu s ohledem na vlastnosti DNA řetězců a nástinu jejich použití pro reálné aplikace dnešní genetiky.

Hlavní rysy všech zmíněných algoritmů jsou přehledně v následující tabulce:

	Přípravná fáze	**Vyhledávací fáze**	**Paměťová náročnost**	**Očekávaná časová složitost vyhledávání**
Brute Force Porovnávání znak po znaku	-	$O(nm)$	-	$O(mn)$
Karp-Rabin Porovnávání čísel namísto znaků, modifikovatelnost	$O(m)$	$O(nm)$	$O(m)$	$O(n+m)$
KMP Posun řízen délkou tzv. "*hranice*"	$O(m)$	$O(nm)$	$O(m)$	$O(n+m)$
Boyer-Moore Chyby *"Špatného znaku"* a *"Správného suffixu"*	$O(m+o)$	$O(nm)$	$O(m+o)$	$O(n/m)$ pro rozsáhlé abecedy
Zhu-Takaoka Chyba *"Špatného znaku"* řízena podle dvojice znaků	$O(m+o^2)$	$O(nm)$	$O(m+o^2)$	$O(n/m)$ i pro malé abecedy
Quick Search Posun podle znaku až za okénkem, modifikovatelnost	$O(m+o)$	$O(nm)$	$O(o)$	$O(n/m)$ pro větší abecedy a krátké vzorové řetězce
Maximal Shift Porovnávání znaků v pořadí podle největšího posunu	$O(m^2+o)$	$O(nm)$	$O(m+o)$	$O(n/m)$ s výjimkou extrémních případů

Tabulka 5.7.1 Shrnutí všech algoritmů

Nejlepší dosaženou časovou složitostí je $O(n/m)$, přičemž tuto mají společnou algoritmy Boyer-Mooreův, Zhu-Takaokův, Quick Search a Maximal Shift. Jednotlivé algoritmy však této složitosti dosahují při různých použitích, díky čemuž se nabízí i použití v DNA aplikacích. Tyto složitosti jsou podloženy i naměřenými hodnotami zachycenými v grafech v této práci. Jednotlivá naměřená data navíc ukazují, že použití zmíněných algoritmů je v reálném životě možné a v programové části je následně i implementované.

Na základě zjištěných a naměřených dat byly v této práci naznačeny i speciální případy, kdy bude použití některého z algoritmů z časového hlediska vhodnější než použití jiného postupu, a mimo svou hlavní oblast použití je tak naznačeno i jejich možné nasazení v menších specificky zaměřených projektech.

Programová část této práce byla navržena striktně s ohledem na Objektově orientované programování, a umožňuje relativně snadné doplnění dalších algoritmů. Použitá struktura programu zpřístupňuje i snadnou modifikaci některých algoritmů pro jejich optimalizaci na konkrétní druh úlohy – zejména se jedná o algoritmy Karp-Rabinův a algoritmus Quick Search.

Definice

abeceda řetězce – všechny textové znaky vyskytující se v daném řetězci.

zdrojový řetězec – řetězec znaků, neboli text, ve kterém se vyhledává.

vzorový řetězec – řetězec znaků, který se vyhledává.

vyhledávací okénko – ohraničení části textu ve zdrojovém řetězci takové, že v jednom kroku může algoritmus porovnávat všechny znaky obsažené ve vyhledávacím okénku.

hash funkce - posloupnost příkazů, s jejichž pomocí lze jednoznačně převést zadaná data na číselo.

prefix a **suffix –** část řetězce obsahující několik prvních, resp. posledních znaků původního řetězce.

hranice – délka nejdelšího prefixu, který je shodný se některým suffixem stejného řetězce.

chyba špatného znaku – výskyt neshody, kdy se porovnávaný znak nevyskytuje ve vzorovém řetězci.

chyba správného suffixu - chyba, kdy se porovnávaný znak ve vzorovém řetězci vyskytuje, jen je na jiné pozici.

n – délka zdrojového řetězce (ve kterém se vyhledává).

m – délka hledaného vzorku (který se vyhledává).

o – počet různých znaků v abecedě.

Literatura

[1] České stránky o genetice

URL: http://genetika.wz.cz (ke dni 22. 4. 2011)

[2] Učebnice Genetiky pro vysoké školy

D. Peter Snustad a Michael J. Simmons: Genetika. Nakladatelství Masarykovy univerzity v Brně, 2009

[3] Informace o DNA

URL: http://cs.wikipedia.org/wiki/DNA (ke dni 22. 4. 2011)

[4] Přehledný slovník pojmů genetiky

URL: http://www.genomia.cz/cz/slovnik-pojmu/ (ke dni 22. 4. 2011)

[5] Informační stránky genetické laboratoře

URL: http://www.neb.com (ke dni 22. 4. 2011)

[6] Informace o DNA

RNDr. Emanuel Žďárský CSc., Osobní konzultace

[7] Databáze DNA řetězců

URL: http://www.ncbi.nlm.nih.gov/

[8]Databáze DNA řetězců

URL: http://genome.ucsc.edu/

[9] Popis algoritmů pro přesnou shodu znaků

Christian Charras, Thierry Lecroq: *Exact String Matching Algorithms*, King's College Publications, 2004

[10] Karp-Rabinův algoritmus - popis

R. M. Karp and M. O. Rabin: "Efficient randomized pattern-matching algorithms," *IBM J. Research and Development*, 1987

[11] IT magazín Reboot.cz – popis algoritmů Knuth-Morris-Pratt a Boyer-Moore

URL: http://reboot.cz/ (ke dni 22. 4. 2011)

[12] Boyer-Mooreův algoritmus - popis

Boyer R.S., Moore J.S., A fast string searching algorithm. *Communications of the ACM*. 20:762-772, 1977

[13] National Institute of Standarts and Technology – popis Zhu-Takaokova algoritmu

URL: http://www.nist.gov/dads/HTML/zhuTakaoka.html (ke dni 22. 4. 2011)

[14] Záznam z konference o vyhledávacích algoritmech

Tuning the Zhu-Takaoka string matching algorithm and experimental results. Kybernetika, vol. 38, issue 1, 2002

[15] Skripta Flensburgské univerzity aplikovaných věd – Quick Search algoritmus

Hans Werner Lang: Algorithmen in Java, 2. Vydání, Flensenburg, 2006

[16] Sundayův Quick Search algoritmus - popis

Daniel M. Sunday: A very fast substring **search algorithm**, Communications of the Association for Computing Machinery, ACM Press New York, NY, USA,

Příloha A

Programátorská dokumentace

Volba programovacího jazyka

K implementaci programu jsem zvolil programovací jazyk Java. Hlavními důvody pro volbu tohoto jazyka byla jednak snadná přenositelnost mezi různými operačními systémy, dále fakt, že v tomto jazyce jsem byl schopný naprogramovat grafické uživatelské rozhraní i nejhlubší obecné znalosti o možnostech jazyka.

Zdrojové kódy

Z důvodu jejich značného rozsahu zde zdrojové kódy nejsou uvedeny a jsou obsaženy pouze na přiloženém CD.

Popis tříd

Spuštění programu vyvolá spuštění třídy `Start`, které slouží pouze pro spuštění programu a zpracování případných parametrů z příkazové řádky.

Třída `Constants` obsahuje veškeré ostatní konfigurační konstanty pro nastavení programu.

Po svém spuštění program inicializuje grafické rozhraní, které dál vytvoří instance akcí `CustomAction` a třídu pro zpracování DNA záznamu `SourceDNAhandler`.

Třída `SourceDNAhandler` ještě neprovádí samotné vyhledávání a slouží pouze k načtení dat ze souboru, případně předzpracování vstupních dat do formátu FASTA (.fa). Tato třída si může

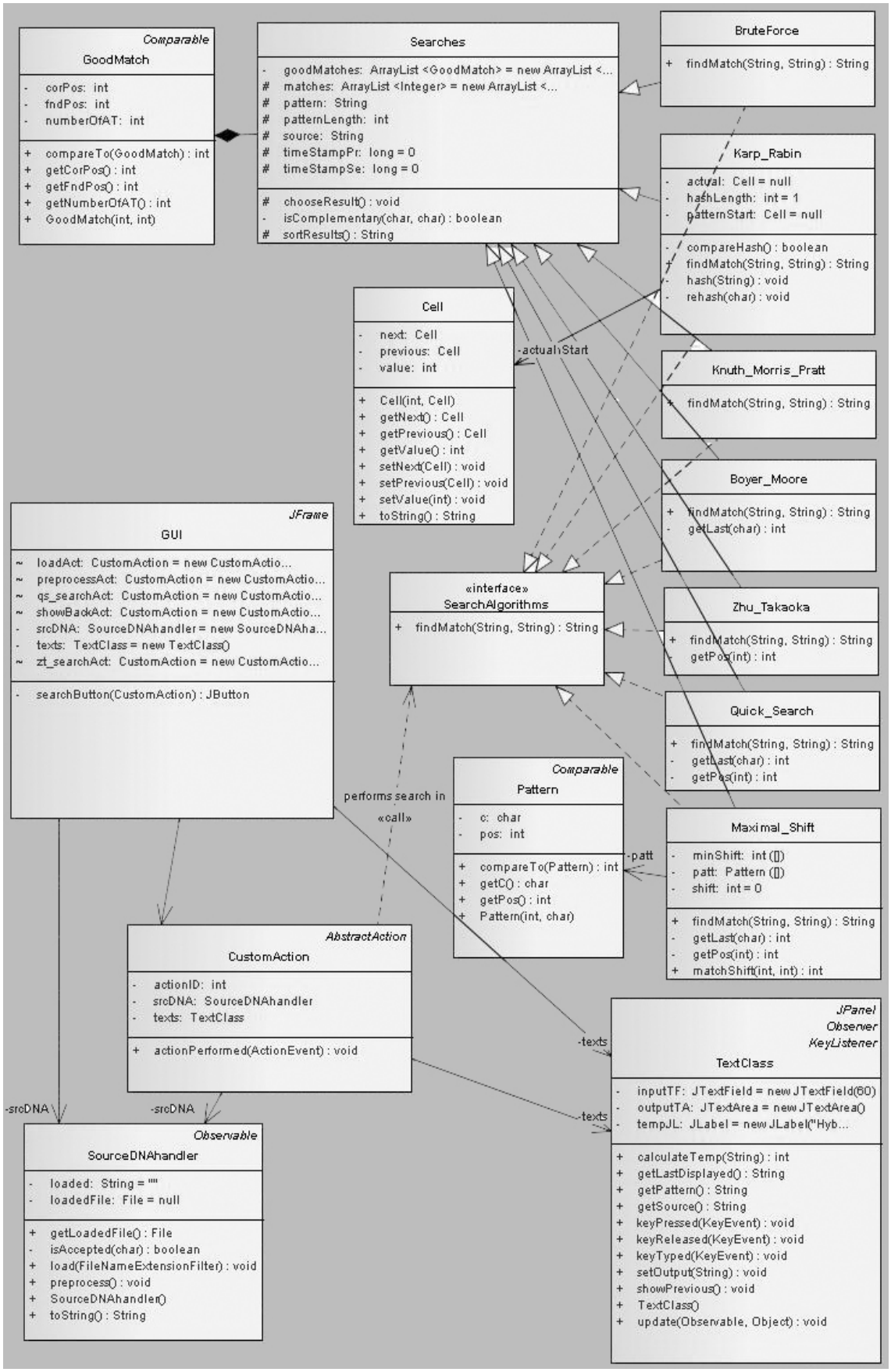
Až při stisku tlačítka pro vyhledávání vytvoří třída `CustomAction` instanci příslušného algoritmu a přes rozhraní `SearchAlgorithms` spustí vlastní vyhledávání.

Třída `Searches`, od které dědí všechny použité algoritmy spravuje kolekci nalezených shod, provádí dohledání odpovídajícího druhého řetězce a sežazení vícenásobných nálezů podle počtu znaků **A** a **T** mezi nalezenými řetězci (kroky 2 a 3 tak jak jsou popsány v kapitole 3.1). Tato třída navíc provádí i měření časové náročnosti algoritmů.

Třída `TextClass` poskytuje prostředky a stará se o prezentaci výsledků uživateli. Kromě samotné prezentace výsledků ještě počítá hybridizační teplotu řetězce zadaného uživatelem.

Uml Diagram

Všechny náležitosti programové části přehledně zachycuje následující UML diagram:



Příloha A - UML Diagram

Příloha B

Uživatelská dokumentace

Instalace programu

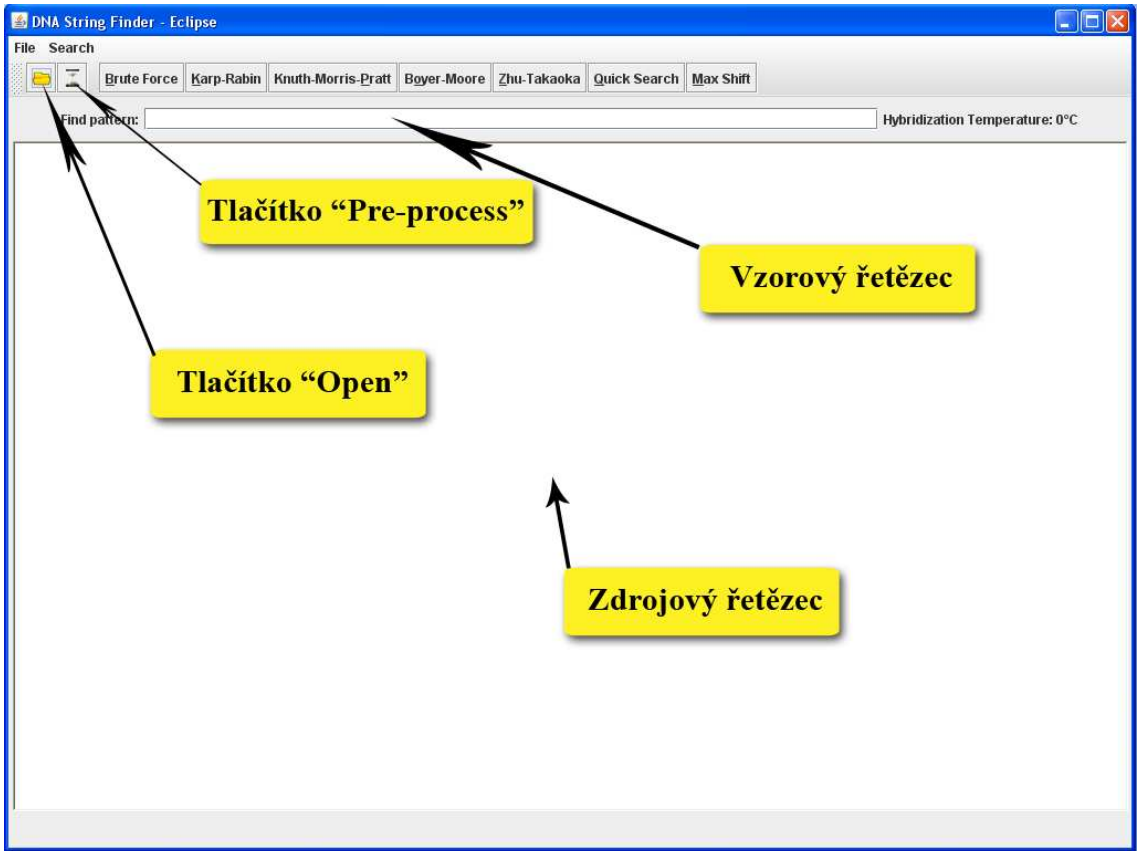
Program samotný nevyžaduje žádnou instalaci. Nicméně vzhledem k použitému programovacímu jazyku je nutné mít nainstalován Java Runtime Environment verze 1.5 nebo vyšší.

Spuštění programu

Program lze spustit spuštěním „DNA-String-Finder.jar“, nebo přes příkazovou řádku za pomoci příkazu „DNA-String-Finder.jar“. Před spuštěním programu je též možné si program redistribuovat za pomoci přiloženého „Build.xml“.

Program má dva volitelné parametry pro spuštění – šířku a výšku okna , které musí být zadány jako celé kladné číslo.

Běh programu



Příloha B - Grafické rozhraní programu

Po spuštění programu je nutné zadat vzorový řetězec do políčka nadepsaného „find pattern“. Napravo od tohoto políčka se automaticky počítatá hybridizační teplotu zadaného řetězce. Dále je nutné zadat i zdrojový řetězec. To lze buď ručně do velké

textové oblasti, nebo pomocí tlačítka „Open“ (V menu File > Open) otevřít soubor obsahující zdrojový řetězec.

Pokud při otevírání zvolíme jiný formát než FASTA, je vhodné před samotným vyhledáváním převést otevřený soubor do tohoto formátu pomocí tlačítka „Pre-process“ (File > Pre-Process). V textovém okně programu se objeví prvních 40 000 znaků načteného souboru.

Následně volbou algoritmu zahájí program vyhledávání.

Položka v menu „Step back“ zobrazí textový obsah předešlého kroku.

Zadání údajů

Pro zadávání obou řetězců je vhodné zadávat jen základní znaky DNA abecedy, tedy **A**, **C**, **G** a **T**. Ostatní znaky budou v současné konfiguraci programu ignorovány. Pro zadávání zdrojového řetězce silně doporučuji použít možnost načtení ze souboru ve formátu FASTA, případně využít i možnosti převedení do toho formátu, čímž se eliminují veškeré neplatné znaky. Samotný proces převedení může trvat delší dobu v závislosti na délce řetězce

Ukončení programu

Program lze vypnout standartním způsobem pomocí křížku v pravém horním rohu programu.

Příloha C

Tabulky naměřených hodnot

Přípravná fáze

	Brute Force	Karp-Rabin	KMP	Boyer-Moore	Zhu-Takaoka	Quick-Search	Max Shift
18	0	6984	4191	5867	21511	7822	28495
19	0	6984	3911	5867	18997	8101	27378
20	0	6984	4190	5308	13410	8102	29054
21	0	6984	3911	6146	13968	8939	25702
22	0	7264	3911	5587	13130	8381	25981
23	0	6705	4191	5308	12013	8660	24026
24	0	6984	3911	6146	13689	8381	23746
25	0	8381	3632	5028	11453	9219	24025
26	0	6891	3911	4750	12851	7263	21511
27	0	6704	3911	5028	11733	9219	24305
28	0	6705	3353	5587	12851	8940	24584

Tabulka C.1 - Tabulka přípravných fází

Brute Force

	cctaaatagttcatcacaTGAGAAGCTA	cagcagtgggggggccgCAGCAGTTTT	aaaaccccggggttttaaACCCGGGTTT	xxxxxxxxxxxxxxxxxxXXXXXXXXX
18	1 986 007	2 125 969	2 131 276	1 343 467
19	1 987 403	2 121 778	2 152 229	1 339 835
20	1 998 578	2 117 867	2 187 429	1 341 511
21	2 060 038	2 215 365	2 229 054	1 344 584
22	2 163 125	2 118 984	2 351 137	1 347 378
23	2 163 962	2 123 175	2 201 956	1 339 276
24	2 161 448	2 161 727	2 350 299	1 354 362
25	2 173 181	2 214 528	2 271 797	1 346 540
26	2 175 975	2 207 543	2 346 946	1 351 569
27	2 198 324	2 253 638	2 324 597	1 373 080
28	2 196 368	2 268 724	2 232 966	1 358 273

Tabulka C.2 - Algoritmus Brute Force

Karp-Rabin

	cctaaatagttcatcacaTGAGAAGCTA	cagcagtgggggggccgCAGCAGTTTT	aaaaccccggggttttaaACCCGGGTTT	xxxxxxxxxxxxxxxxxxXXXXXXXXX
18	1 848 280	1 900 241	1 831 518	1 777 042
19	1 840 458	1 912 813	1 855 264	1 794 083
20	1 844 927	1 913 651	1 801 346	1 789 613
21	1 844 089	1 912 254	1 804 140	1 774 807
22	1 847 442	1 937 956	1 826 210	1 776 762
23	1 864 483	1 920 915	1 795 479	1 764 191
24	1 872 026	1 922 927	1 790 730	1 789 613
25	1 862 248	1 923 708	1 823 416	1 794 921
26	1 856 381	1 936 838	1 819 505	1 772 292
27	1 854 146	1 870 980	1 853 867	1 791 848
28	1 924 546	1 906 109	1 836 267	1 777 600

Tabulka C.3 - Karp-Rabinův algoritmus

Knuth-Morris-Pratt

	cctaaatagttcatcacaT GAGAAGCTA	cagcagtgggggggccgC AGCAGTTTT	aaaaccccggggttttaaA CCCGGGTTT	xxxxxxxxxxxxxxxxxxX XXXXXXXXX
18	2 133 511	2 323 759	2 018 413	1 734 857
19	2 137 981	2 392 483	2 018 134	1 730 388
20	2 136 864	2 448 915	2 090 210	1 709 715
21	2 195 530	2 317 333	2 016 737	1 710 274
22	2 130 438	2 399 467	2 039 644	1 729 829
23	2 130 718	2 449 753	2 015 340	1 713 905
24	2 182 121	2 372 927	2 039 365	1 716 140
25	2 210 895	2 415 391	2 039 924	1 707 480
26	2 211 454	2 450 312	2 057 524	1 713 905
27	2 097 473	2 451 209	2 153 906	1 782 070
28	2 212 851	2 377 397	2 252 801	1 714 184

Tabulka C.4 - Knutk-Morris-Prattův algoritmus

Boyer-Moore

	cctaaatagttcatcacaT GAGAAGCTA	cagcagtgggggggccgC AGCAGTTTT	aaaaccccggggttttaaA CCCGGGTTT	xxxxxxxxxxxxxxxxxxX XXXXXXXXX
18	1 775 924	2 634 431	1 756 089	268 470
19	1 533 435	3 353 778	1 676 749	253 664
20	1 366 654	1 069 969	1 338 438	239 975
21	1 768 381	1 671 162	1 827 886	231 873
22	1 858 616	2 818 235	1 628 699	230 477
23	2 120 660	1 033 930	1 561 092	212 038
24	1 520 025	1 437 334	1 868 394	202 540
25	1 880 128	933 359	1 660 826	196 973
26	1 117 181	1 685 131	1 205 460	191 086
27	1 101 816	1 651 886	1 809 168	183 264
28	2 041 041	1 316 368	1 748 267	178 515

Tabulka C.5 - Boyer-Mooreův algoritmus

Zhu-Takaoka

	cctaaatagttcatcacaT GAGAAGCTA	cagcagtgggggggccgC AGCAGTTTT	aaaaccccggggttttaaA CCCGGGTTT	xxxxxxxxxxxxxxxxxxX XXXXXXXXX
18	894 527	1 158 248	1 038 121	213 993
19	939 785	1 105 447	978 336	201 422
20	870 502	803 454	1 096 229	191 365
21	811 276	1 014 096	896 762	186 057
22	811 276	1 172 774	842 007	177 956
23	755 124	810 997	738 921	168 457
24	777 194	928 889	729 423	160 007
25	980 572	893 689	706 514	151 975
26	807 924	894 807	626 337	150 858
27	822 451	998 451	648 686	145 270
28	854 578	904 864	765 460	139 683

Tabulka C.6 - Zhu-Takaokův algoritmus

Quick Search

	cctaaatagttcatcacaT GAGAAGCTA	cagcagtgggggggccgC AGCAGTTTT	aaaaccccggggttttaaA CCCGGGTTT	xxxxxxxxxxxxxxxxxxX XXXXXXXXX
18	1 378 108	771 047	1 250 159	271 264
19	1 544 051	662 096	1 232 838	258 692
20	2 235 479	1 208 813	1 602 438	248 355
21	2 152 787	1 144 000	1 234 794	240 254
22	1 851 912	961 574	1 075 556	227 124
23	1 769 220	1 057 956	1 376 153	220 699
24	1 363 861	1 048 178	1 164 115	211 200
25	1 256 304	2 576 864	1 019 403	204 495
26	1 481 194	1 903 035	1 730 667	199 746
27	2 538 032	1 664 457	1 368 889	194 159
28	2 503 091	1 380 901	1 232 559	186 616

Tabulka C.7 - Quick Search algoritmus

Maximal Shift

	cctaaatagttcatcacaT GAGAAGCTA	cagcagtgggggggccgC AGCAGTTTT	aaaaccccggggttttaaA CCCGGGTTT	xxxxxxxxxxxxxxxxxxX XXXXXXXXX
18	977 499	624 661	784 736	276 572
19	1 046 502	586 108	750 933	265 118
20	779 150	582 476	878 324	253 384
21	865 752	591 695	857 930	243 885
22	806 528	563 479	776 076	234 108
23	786 413	623 543	869 664	224 609
24	751 772	626 337	811 835	216 228
25	733 613	606 781	757 962	208 685
26	809 880	685 004	878 883	202 820
27	958 781	661 257	834 743	198 629
28	954 870	646 730	759 593	198 908

Tabulka C.8 - Maximal Shift algoritmus