VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2017 Bc. Lukáš Klouba

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV BIOMEDICÍNSKÉHO INŽENÝRSTVÍ
DEPARTMENT OF BIOMEDICAL ENGINEERING

CELOGENOMOVÉ ZAROVNÁNÍ POMOCÍ SUFFIXOVÝCH STROMŮ
WHOLE GENOME ALIGNMENT USING SUFFIX TREES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. Lukáš Klouba
AUTHOR

VEDOUCÍ PRÁCE Ing. Denisa Maděránková, Ph.D.
SUPERVISOR

BRNO 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Biomedicínské inženýrství a bioinformatika**
Ústav biomedicínského inženýrství

Student: Bc. Lukáš Klouba ***ID:*** 155583
Ročník: 2 ***Akademický rok:*** 2016/17

NÁZEV TÉMATU:

Celogenomové zarovnání pomocí suffixových stromů

POKYNY PRO VYPRACOVÁNÍ:

1) Vypracujte literární rešerši na téma algoritmy pro konstrukci suffixových matic a stromů. 2) Dále vypracujte rešerši metod celogenomového zarovnání. 3) V programovém prostředí R realizujte algoritmus na tvorbu suffixového stromu s lineární časovou náročností. 4) Navrhněte a realizujte metodu zarovnání genomů porovnáním jejich suffixových stromů. 5) Zarovnávací algoritmus otestujte na reálných bakteriálních genomech. 6) Výsledky diskutujte a porovnejte s výsledky zarovnání např. nástrojem Clustal Omega.

DOPORUČENÁ LITERATURA:

[1] UKKONEN, E. On-line construction of suffix trees. Algorithmica. 1995, 3, 249-260.

[2] DELCHER, A.L.; PHILLIPPY, A.; CARLTON, J.; SALZBERG, S.L. Fast algorithm for large-scale genome alignment and comparison. Nucleic Acids Research. 2002, 30 (11), 2478-2483.

Termín zadání: 6.2.2017 ***Termín odevzdání:*** 19.5.2017

Vedoucí práce: Ing. Denisa Maděránková, Ph.D.
Konzultant:

prof. Ing. Ivo Provazník, Ph.D.
předseda oborové rady

UPOZORNĚNÍ:
Autor diplomové práce nesmí při vytváření diplomové práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

ABSTRAKT

Cílem této práce je vytvořit algoritmus, který pomocí sufixových konstrukcí umožní zarovnání genomu dvou organismů, a implementovat jej do programového prostředí jazyka R. Práce se věnuje popisu konstrukce sufixových struktur a metodami celogenomového zarovnání. Výsledkem práce je funkční algoritmus pro celogenomové zarovnání pomocí sufixových struktur implementovaný v programovém prostředí R a jeho srovnání s podobnými programy pro celogenomové zarovnání.

KLÍČOVÁ SLOVA

sufixové stromy, sufixová pole, Ukkonenův algoritmus, R, celogenomové zarovnání

ABSTRACT

The aim of this thesis is to create an algorithm that allows the alignment of the genome of two organisms by means of suffix structures and to implement it into the programming language environment R. The thesis deals with the description of the construction of the suffix structures and the methods of whole genome alignment. The result of the thesis is a functional algorithm for whole genome alignment by means of suffix structures implemented in the software environment R and its comparison with similar programs for the whole genome alignment.

KEYWORDS

suffix tree, suffix array, Ukkonen's algorithm, R, whole-genome alignment

KLOUBA, Lukáš *Celogenomové zarovnání pomocí suffixových stromů*: diplomová práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav biomedicínského inženýrství, 2017. 50 s. Vedoucí práce byl Ing. Denisa Maděránková, Ph.D.

Vysázeno pomocí balíčku `thesis` verze 2.61; `http://latex.feec.vutbr.cz`

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Celogenomové zarovnání pomocí suffixových stromů“ jsem vypracoval(a) samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor(ka) uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil(a) autorská práva třetích osob, zejména jsem nezasáhl(a) nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom(a) následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora(-ky)

OBSAH

1 Úvod **1**

2 Sufixové stromy **3**

2.1 Tvorba sufixového stromu 4

2.2 Využití sufixových struktur 7

2.2.1 Příklad vyhledávání pomocí sufixového stromu 7

3 Sufixová pole **9**

3.1 Obecná tvorba sufixové matice 10

3.2 Sufixové pole - nejjednodušší řešení 10

3.2.1 Princip 10

3.2.2 Zhodnocení 10

3.3 Sufixové pole - algoritmus Manber & Myers 11

3.3.1 Princip algoritmu Manber & Myers 11

3.3.2 Zhodnocení algoritmu Manber & Myers 14

3.4 Sufixové pole řazené pomocí algoritmu SA-IS 14

3.4.1 Princip výpočtu 14

3.4.2 Zhodnocení algoritmu SA-IS 17

3.5 Vyhledávání v sufixovém poli 18

3.6 Nalezení shod dvou různých vstupů pomocí sufixového pole 18

4 Celogenomové zarovnání **19**

4.1 Needlemanův-Wunschův algoritmus 19

4.1.1 Princip algoritmu 19

4.1.2 Zhodnocení algoritmu 21

4.2 Smithův-Watermanův algoritmus 21

4.2.1 Princip algoritmu 21

4.2.2 Zhodnocení algoritmu 22

4.3 Program Clustal Omega 22

4.3.1 Výstup z programu 23

4.3.2 Zhodnocení programu 24

4.4 Celkové zhodnocení 24

5 Program **25**

5.1 Programovací jazyk R 25

5.2 Tvorba sufixových stromů 25
5.2.1 Zhodnocení programu pro tvorbu sufixových stromů 28
5.3 Tvorba sufixových polí 28
5.4 Celogenomové zarovnání 31
5.5 Výstupy z programu 31
5.5.1 Hledání mutovaného genu v originální sadě dat 32
5.5.2 Hledání referenčního genu v celém genomu 36
5.5.3 Zarovnání dvou genomů 37
5.6 Porovnání s jinými zarovnávacími algoritmy 40
5.6.1 On-line zarovnávací nástroje 40
5.6.2 Program MUMmer 41
5.6.3 Zhodnocení 43

6 Závěr **44**

Literatura **46**

Seznam symbolů, veličin a zkratek **49**

SEZNAM OBRÁZKŮ

2.1 vpravo: Sufixový strom pro sekvenci AA bez ukončovacího znaku; vlevo: Sufixový strom pro sekvenci $AA\$$ s ukončovacím znakem . . . 3
2.2 Sufixový strom pro sekvenci $GCATCAT\$$ vytvořený Ukkonenovým algoritmem 4
2.3 Vývojový diagram reprezentující Ukkonenův algoritmus 6
2.4 Příklad vyhledávání symbolu A v sufixovém stromě, který je vytvořený pro vstupní sekvenci $ACTGACAC\$$ 7
2.5 Příklad vyhledávání řetězce CTG v sufixovém stromě, který je vytvořený pro vstupní sekvenci $ACTGACAC\$$ 8
2.6 Příklad vyhledávání řetězce $ACAT$ v sufixovém stromě, který je vytvořený pro vstupní sekvenci $ACTGACAC\$$ 8
3.1 Vývojový diagram algoritmu Manber & Myers 13
4.1 Blokové schéma principu fungování programu Clustal Omega. 23
5.1 Vývojový diagram zjednodušeně popisující funkci navrženého programu na tvorbu sufixového stromu. 26
5.2 Vývojový diagram části programu na vytvoření sufixového pole 30
5.3 Graf ukazující výstup z programu pro 1% mutace při minimální shodě 20 a maximální mezeře 20 33
5.4 Graf ukazující výstup z programu pro 5% mutace při minimální shodě 20 a maximální mezeře 20 33
5.5 Graf ukazující výstup z programu pro 5% mutace při minimální shodě 10 a maximální mezeře 20 34
5.6 Graf ukazující výstup z programu pro 10% mutace při minimální shodě 20 a maximální mezeře 20 34
5.7 Graf ukazující výstup z programu pro 10% mutace při minimální shodě 5 a maximální mezeře 20 35
5.8 Graf ukazující výstup z programu pro 10% mutace při minimální shodě 5 a maximální mezeře 30 35
5.9 Graf ukazující výstup z programu pro zarovnání sekvence genu dnaQ z genomu SamoaD s celým genomem Chicago 37
5.10 Graf ukazující výstup z programu pro zarovnání části genomu SamoaD [1:10] se stejně velkou částí genomu Chicago 38
5.11 Graf ukazující výstup z programu pro zarovnání části genomu SamoaD [1:50] se stejně velkou částí genomu Chicago 39

5.12 Ukázkový graf znázorňující výstup z programu MUMmer jako dot-plot 42

SEZNAM TABULEK

3.1 Příklad sufixového pole pro sekvenci GCATCAT#. 9
3.2 Hledání řetězce CATC ve vstupní sekvenci GCATCAT# 18
4.1 Výchozí tabulka pro Needlemanův-Wunschův algoritmus pro vstupní nukleotidové řetězce ATACGTC a AACTTCA a penalizaci mezery -1. 20
4.2 Tabulka vytvořená Needleman-Wunsch algoritmem pro vstupní nukleotidové řetězce ATACGTC a AACTTCA s penalizaci mezery -1, ohodnocením shody 1 a neshody -1. Zpětná cesta vyznačena tučně. . 20
4.3 Tabulka vytvořená Smith-Waterman algoritmem pro vstupní nukleotidové řetězce AACCTATAGCT a GCGATATA s penalizaci mezery -1, ohodnocením shody 1 a neshody -1. Zpětná cesta vyznačena šipkami.[11] 22
5.1 Nalezené nejdelší shody pro různé procento mutace 32
5.2 Výsledky z celogenomového zarovnání referenčního genu 36
5.3 Výsledky ze zarovnání části genomů 40

1 ÚVOD

V dnešním světě, zaměřeném na technologie, je problém rychle vyhledávat v dlouhých textových řetězcích, jako jsou knihy, blogové články nebo genové sekvence. Při vyhledávání daného řetězce v dlouhém textu bychom ztratili velké množství času. Proto se tato práce zabývá převáděním textových řetězců do podoby sufixových struktur, neboť čas hledání dokáží zkrátit.

Samotným cílem práce je vytvořit program v programovém prostředí R, který pracuje se sufixovými strukturami a jeho účelem je zarovnání celého genomu organismů. Jádrem programu bude vlastní algoritmus, který vytvoří sufixovou strukturu za pomoci postupného třídění. Dále má program využívat pomocné datové struktury pro zrychlení hledání shodných úseků v genetických datech. Program bude využívat buď sufixových stromů nebo sufixových polí.

Sufixový strom je taková datová struktura, která obsahuje všechny přípony (sufixy) daného textového řetězce. Celý strom vychází z hlavního kořene, ze kterého vystupují listy. Listy mohou být zakončeny uzly, ze kterých mohou růst další listy a tak dále.

Sufixové pole je datová struktura připomínající tabulku. Hodnoty v této tabulce udávají indexy seřazených sufixů vstupního řetězce. Oproti sufixovým stromům se tak jedná o strukturu zabírající menší úložný prostor a jejich interpretace je jednodušší a jasnější.

V bioinformatice jsou tyto datové struktury důležité hlavně z důvodu rychlosti. Sufixový strom i sufixové pole je možné zkonstruovat v lineárním čase, což je jediná možnost, jak pracovat se sekvencemi o velikosti několika milionů nukleotidů a nestrávit výpočtem několik let. Hlavní využití je v oblasti zarovnávání sekvencí. Příkladem může být program MUMmer 3.0 který zvládne s pomocí sufixových stromů vypočítat zarovnání u datových souborů o velikosti několika stovek MB během pouhých desítek vteřin.

Bioinformatika se zabývá nejen získáváním a shromažďováním biologických dat, ale také jejich analýzou. K tomu účelu slouží i vícenásobné zarovnání, které umožňuje správně pochopit vlastnosti a účel biologických sekvencí porovnáváním získaných informací. Lze tak pochopit i vztahy mezi na první pohled nesouvisejícími sekvencemi.

V teoretické části se seznámíme s datovou strukturou sufixový strom a podrobně si vysvětlíme princip jeho konstrukce pomocí algoritmu navrženého E. Ukkonenem. Následně se seznámíme s další sufixovou strukturou, kterou je sufixové pole, popíšeme si některé z algoritmů pro jeho konstrukci a zhodnotíme jejich vhodnost pro

implementaci do našeho programu. Dále si vysvětlíme principy metod využívaných pro zarovnávání genetických dat a shrneme klady a zápory popisovaných metod.

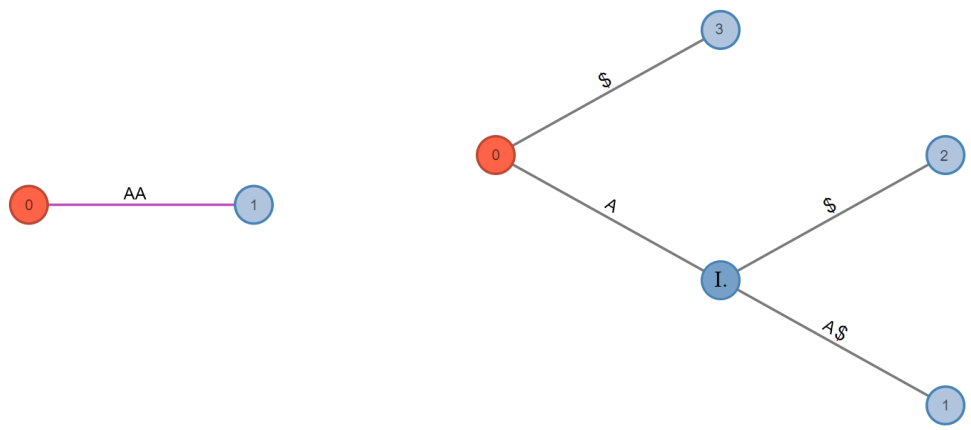
Teoretický rámec této práce vychází především z článků uveřejněných v odborných publikacích a vědeckých časopisech.

Praktická část práce popisuje samotnou implementaci programu na zarovnávání dat postavenou na sufixové struktuře. Ukážeme si nevhodnou implementaci, která by byla použitelná v jiném programovém prostředí a vysvětlíme si, jak funguje výsledné řešení. Nakonec toto řešení porovnáme s již existujícími programy.

2 SUFIXOVÉ STROMY

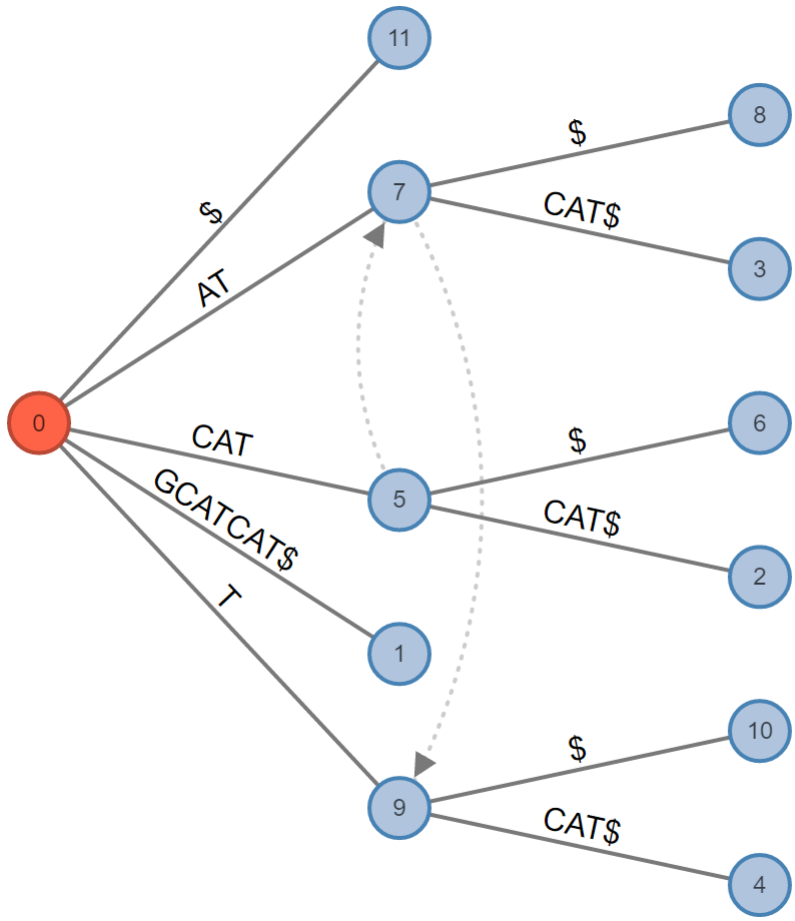
Sufixová pole a sufixové stromy jsou důležitou součástí velkého množství vyhledávacích algoritmů. Hlavním důvodem, proč tomu tak je, je rychlost jejich tvorby, která může dosáhnout až lineární časové náročnosti *O(n)*, kde n je délka vstupního řetězce.[1][2][3]

Obě sufixové struktury jsou vytvářeny ze vstupního řetězce, který je potřeba před začátkem tvorby sufixového stromu i sufixového pole rozšířit o jedinečný symbol, neboli ukončovací znak, který se ve vstupním řetězci nevyskytuje. Je to z toho důvodu, že kdybychom končili existujícím znakem, mohlo by se stát, že tento znak zmizí z výsledného řešení (viz. obr. 2.1). V této práci jsou využívány dva zástupné symboly. Jedním je $, používaný u sufixových stromů a druhým je #, používaná u sufixových polí. Tento výběr je čistě náhodný a nemá žádný konkrétní význam.



Obr. 2.1: vpravo: Sufixový strom pro sekvenci *AA* bez ukončovacího znaku; vlevo: Sufixový strom pro sekvenci *AA$* s ukončovacím znakem

Sufixový strom je datová struktura, která uchovává všechny přípony (sufixy) textového řetězce. Tyto struktury slouží například k rychlému vyhledávání nebo k vícenásobnému zarovnávání genetických sekvencí.



Obr. 2.2: Sufixový strom pro sekvenci *GCATCAT$* vytvořený Ukkonenovým algoritmem

2.1 Tvorba sufixového stromu

Sufixový strom lze vytvořit několika metodami. Nejčastěji používanou metodu tvorby sufixového stromu vymyslel v roce 1995 Esko Ukkonen. Tato metoda je výhodná hlavně proto, že jí je možné zkonstruovat sufixový strom v lineárním čase. To znamená, že doba výpočtu je přímo úměrná délce vstupního řetězce.[3]

V základu algoritmu si definujeme primární kořen, ze kterého budeme strom stavět. Poté v jediném for cyklu procházíme textový řetězec od začátku až do konce. V každém kroku tohoto cyklu řešíme 3 prodlužovací pravidla a jedním z nich se řídíme.

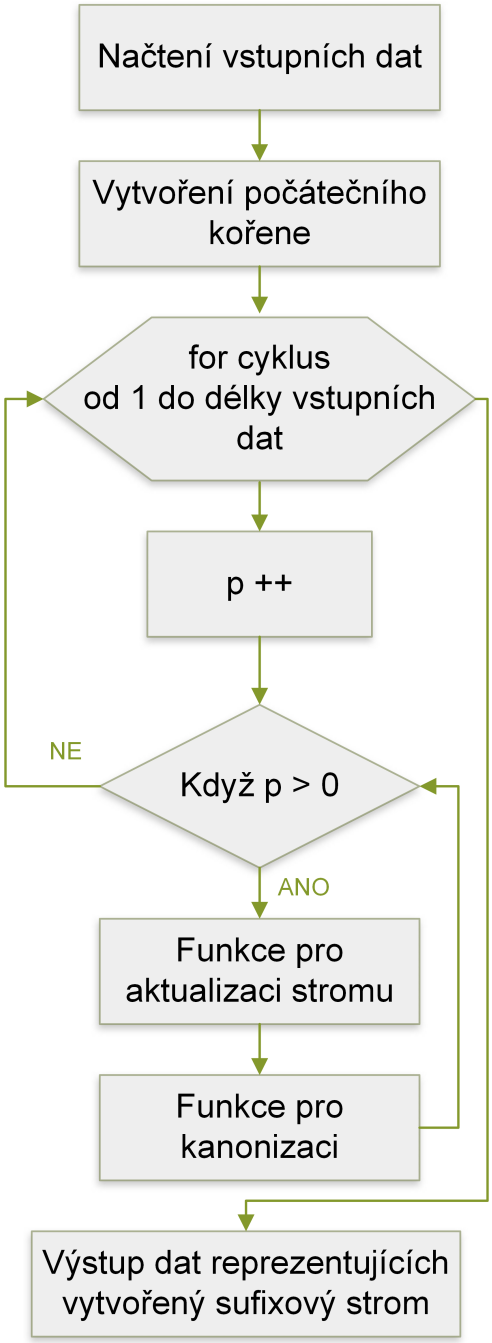
1. Pravidlo 1: V případě, že cesta z označeného kořene končí v okraji listu (tj. S[i] je poslední znak na listu), pak je nutno přidat znak S[i+1] na konec tohoto

listu.

2. Pravidlo 2: Jestliže cesta z označeného kořene končí před okrajem listu (tj. na listu existuje více znaků po znaku S[i]) a další znak není S[i+1], poté je nutno vytvořit nový list začínající symbolem S[i+1].

   Nový vnitřní uzel bude nutno rovněž vytvořit, pokud S[i] končí uvnitř listu (mezi uzly).

3. Pravidlo 3: Pokud cesta z označeného kořene končí před okrajem listu (tj. na listu existuje více znaků po znaku S[i]) a další znak je S[i+1] (list pokračuje stejným znakem, jako vstupní textový řetězec na pozici i+1), poté musíme tento for cyklus ukončit a pokračovat dalším.



Obr. 2.3: Vývojový diagram reprezentující Ukkonenův algoritmus

Na obrázku vývojového diagramu 2.3 můžeme vidět zjednodušené vysvětlení Ukkonenova algoritmu pro konstrukci sufixového stromu. Blok *Funkce pro aktualizaci stromu* obsahuje proceduru pro vytvoření nové větve. Blok *Funkce pro kanonizaci* kontroluje, zda vzdálenost mezi uzly na procházeném listu není menší, než požadovaná procházená délka.

2.2 Využití sufixových struktur

Hlavním důvodem konstrukce sufixových matic a sufixových stromů je potřeba urychlení vyhledávání v dlouhých textových řetězcích, například na webových stránkách nebo v digitalizovaných knihách. Lze pro ně ovšem najít i jiná využití, jako je hledání palindromů nebo zarovnávání sekvencí.

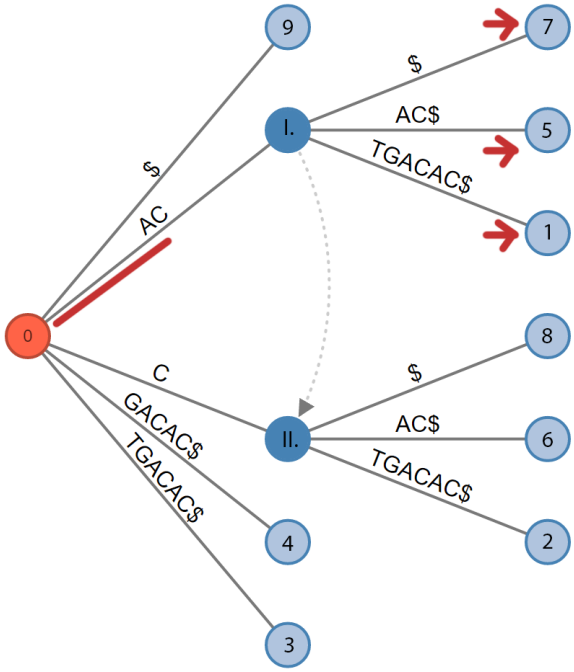
2.2.1 Příklad vyhledávání pomocí sufixového stromu

Předpokládejme, že máme vytvořený sufixový strom ze sekvence *ACTGACAC$*.

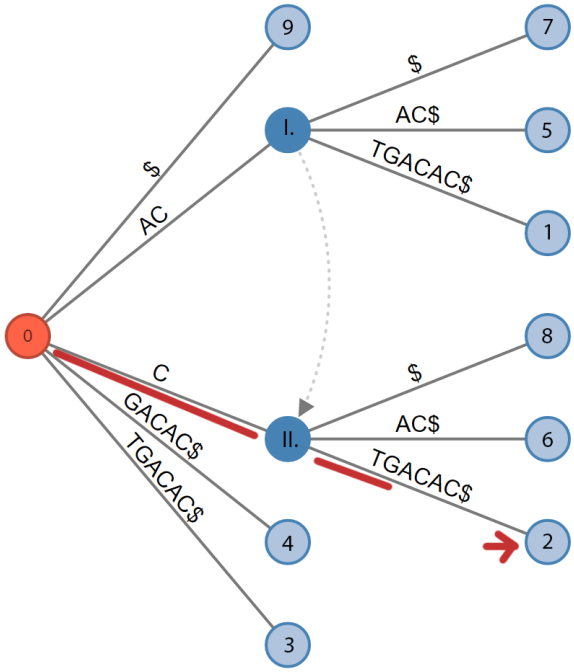
Pokud začneme hledat symbol *A* dostaneme se na větev označenou na obrázku 2.4. Ta pokračuje do uzlu *I.* a uzel se větví na další uzly s indexy 1, 5 a 7. Zjistíme tak, že všechny podřetězce začínající symbolem *A* leží na indexu 1, 5 a 7.

Při hledání řetězce *CTG* projdeme až za uzel *II.* a dojdeme k indexu 2, který nám prozrazuje, že hledaná sekvence leží na druhé pozici ve vstupních datech. (viz. obr. 2.5)

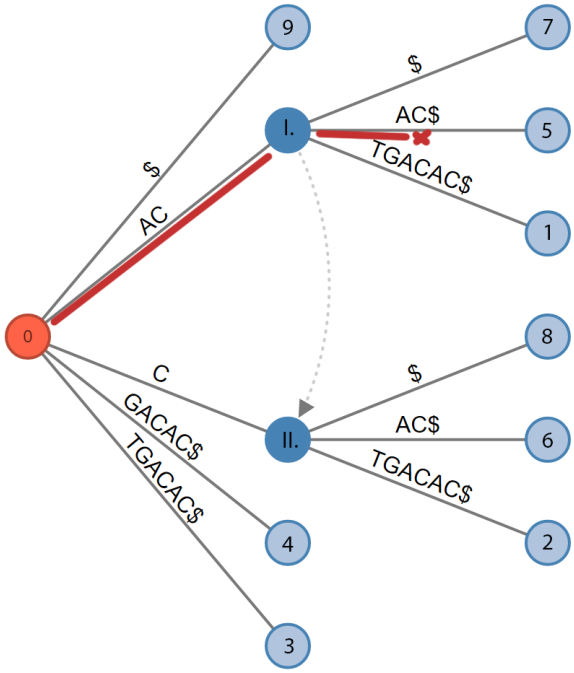
Když budeme chtít hledat například sekvenci *ACAT* zjistíme, že tato sekvence ve vstupních datech neexistuje. (viz. obr. 2.6)



Obr. 2.4: Příklad vyhledávání symbolu *A* v sufixovém stromě, který je vytvořený pro vstupní sekvenci *ACTGACAC$*



Obr. 2.5: Příklad vyhledávání řetězce CTG v sufixovém stromě, který je vytvořený pro vstupní sekvenci $ACTGACAC\$$



Obr. 2.6: Příklad vyhledávání řetězce $ACAT$ v sufixovém stromě, který je vytvořený pro vstupní sekvenci $ACTGACAC\$$

3 SUFIXOVÁ POLE

Sufixový strom je kompaktní konstrukce všech sufixů daného řetězce. Jedná se o silnou datovou strukturu s využitími v numerických aplikacích, bioinformatice a v mnoha dalších vědních oborech. Je možné ji sestavit v lineárním čase závislém na vstupním řetězci.

Sufixové pole je oproti tomu taková datová struktura, odpovídající poli nebo tabulce, ve které najdeme lexikograficky poskládány všechny sufixy vstupního řetězce, popřípadě jejich indexy. Kromě toho jsou v ní obsaženy i všechny další informace, jako v sufixovém stromě, jen v jasnější, jednodušší a kompaktnější formě.

Sufixové pole vzniklo hlavně z důvodu prostorové náročnosti sufixových stromů. Jeho paměťová úspora byla při představení v roce 1990 až 5 krát větší oproti sufixovému stromu, jehož paměťové nároky byly i 27 krát větší, než délka vstupního řetězce.[4]

V dnešní době existují algoritmy, které dokáží sestrojit sufixový strom v prostoru zabírajícím 8 až 14 násobek velikosti vstupního řetězce. Výpočet sufixového pole pomocí algoritmu Manber & Myers zabere v paměti pouze 5 krát délku vstupního řetězce a novější algoritmy pro výpočet sufixových polí dokáží paměťový prostor využít ještě efektivněji.

Tab. 3.1: Příklad sufixového pole pro sekvenci GCATCAT#.

index	Suffixy	SA	ISA	Seřazeno	LCP
1	GCTACTA#	8	6	#	-1
2	CTACTA#	6	5	AT#	0
3	TACTA#	3	3	ATCAT#	2
4	ACTA#	5	8	CAT#	0
5	CTA#	2	4	CATCAT#	3
6	TA#	1	2	GCATCAT#	0
7	A#	7	7	T#	0
8	#	4	1	TCAT#	1

V této kapitole si popíšeme některé algoritmy konstrukce sufixového pole, které jsem se pokusil implementovat do programového jazyku R, ale které měly velmi vysoké nároky na prostor nebo je nebylo možné použít z důvodu vysoké časové náročnosti v tomto programovacím prostředí.

3.1 Obecná tvorba sufixové matice

Nejprve je nutné definovat, co přesně sufixové pole je a jak vzniká. Dále si uvedeme některé pomocné datové struktury, které byly ve výsledném zarovnávajícím programu použity.

Na uvedeném příkladu sufixové matice v tabulce 3 můžeme vidět ve druhém sloupečku všechny vypsané sufixy a v pátém sloupečku všechny sufixy seřazené abecedně.

První sloupec udává index daného sufixu. To znamená pořadí prvku ve vstupních datech, od kterého je sufix tvořen.

Sloupec SA (suffix array, sufixové pole) odpovídá seřazené posloupnosti jednotlivých sufixů. Je vytvořen z indexů jednotlivých sufixů, které jsou seřazeny abecedně.

ISA (inverse suffix array, inverzní sufixová matice) je pomocná datová struktura využívaná v mnoha algoritmech. Je definována jako

$$ISA[i] = j <=> SA[j] = i$$

Další pomocnou datovou strukturou, využívanou hlavně pro vyhledávání, je LCP (longest common prefix, nejdelší společná předpona). Jedná se o nejdelší společný prefix setříděných sufixů. Tato hodnota se používá hlavně k určení podobnosti mezi dvěma řetězci. [5][2]

3.2 Sufixové pole - nejjednodušší řešení

Jako nejjednodušší řešení lze považovat takový přístup, kdy ze vstupního řetězce vytvoříme samostatné sufixy, které následně setřídíme libovolnou řadící funkcí.

3.2.1 Princip

Ze vstupního textového řetězce vytvoříme pole, které bude obsahovat všechny sufixy tohoto řetězce. Toto pole vložíme do třídící funkce, která nám vrátí indexy střižených sufixů vstupního textového řetězce. Tyto indexy odpovídají sufixovému poli SA.

3.2.2 Zhodnocení

Algoritmus je velmi jednoduchý jak na pochopení, tak na zkonstruování. Jeho hlavními nevýhodami jsou časová složitost a paměťová náročnost. Časová složitost

je závislá hlavně na použitém třídícím algoritmu. V nejhorším případě může být u některých třídících funkcí časová náročnost až $O(n^3)$. Prostorová složitost bude $O(n^2)$, protože pro setřídění si musíme v paměti uchovat každý sufix vstupního textového řetězce. V obou případech je n délka vstupního řetězce.

3.3 Sufixové pole - algoritmus Manber & Myers

Sufixové pole bylo představeno světu v roce 1990 Manberem a Myersem. Ti jej definovali jako jednoduchou a prostorově efektivní alternativu za sufixové stromy a navrhly algoritmus pro jeho konstrukci. Tento algoritmus nebyl rychlejší, než algoritmus pro tvorbu sufixového stromu, ale jeho prostorová náročnost byla až 3× menší.[4]

3.3.1 Princip algoritmu Manber & Myers

Jedná se o rekurzivní algoritmus, který je založen na iterativním zpřesňováním sufixového pole. První sufixové pole je získáno pomocí setřídění prvních znaků každého sufixu a následným získáním prvních pozic rozdílných symbolů.

Výpis 3.1: Příklad vytvoření inicializačního pole SA1

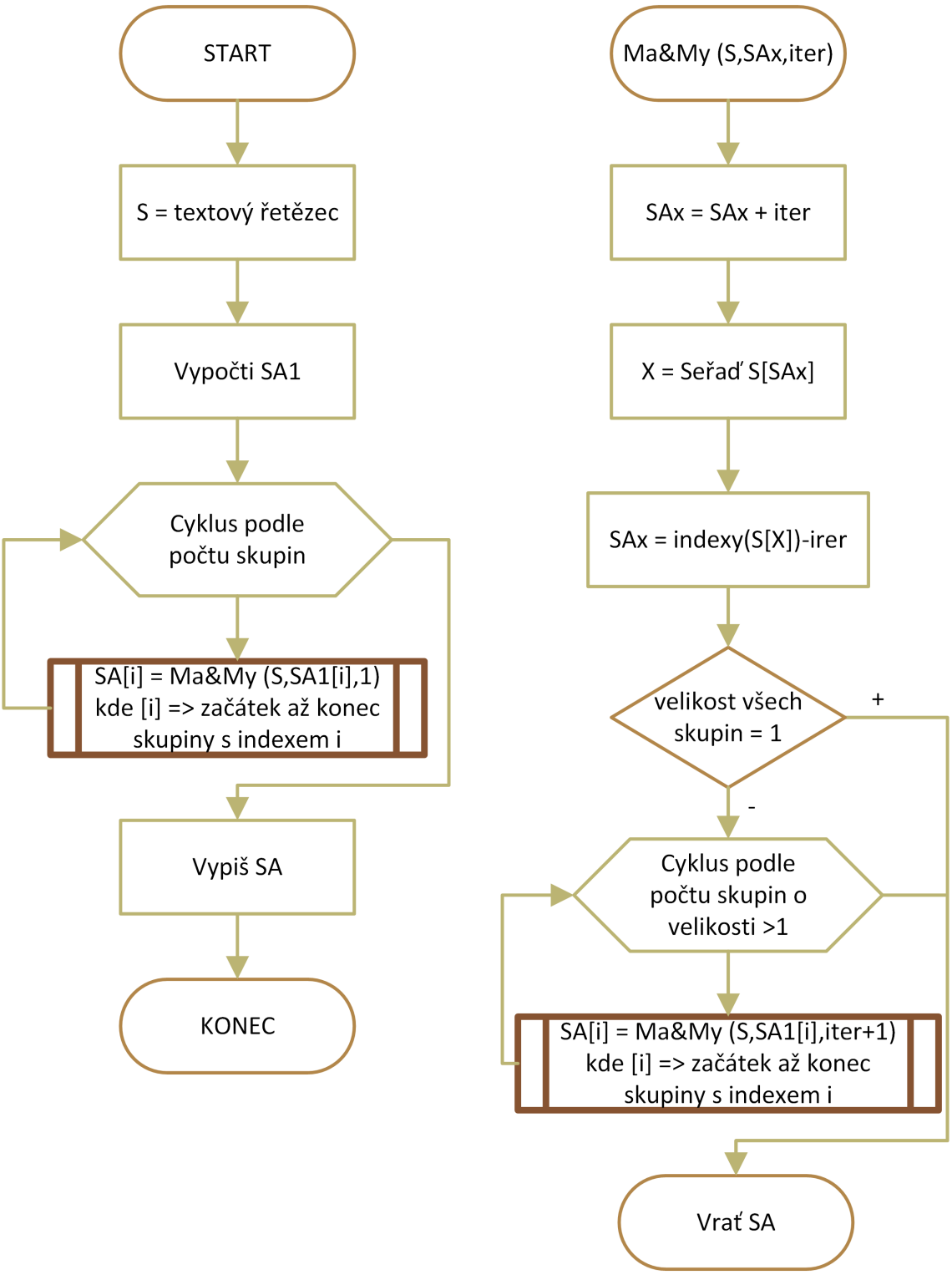
```
Vstupní sekvence:
A T C T G A C G A
Indexy jednotlivých symbolů:
1 2 3 4 5 6 7 8 9
Setříděné první symboly všech sufixů:
A A A/C C/G G/T T
Inicializační SA - indexy setříděných symbolů:
1 6 9 3 7 5 8 2 4
```

Ve výpisu 3.1 můžeme vidět jednotlivé rozdělení seřazených symbolů do sekcí stejných symbolů naznačené lomítky. Každá tato skupina vstupuje do další iterace algoritmu a dále se třídí. Třídění v dalších iteracích je prováděno tak, že vezmeme vstupní indexy a přičteme k nim hloubku iterace (v první iteraci jedničku, v další iteraci dvojku, atd.). Na takto získaných indexech zjistíme symboly ze vstupního řetězce, a ty pak seřadíme. Pokud vznikne alespoň jedna skupina s více, než jedním znakem, postupuje tato skupina do další iterace, dokud počet symbolů každé skupiny dané iterace není roven jedné.

Výpis 3.2: Příklad první iterace pro první skupinu z předchozího příkladu

```
Indexy z první skupiny z příkladu výše:
1 6 9
Přičtení hloubky iterace:
2 7 10
Symboly na daných pozicích:
T C Na
Seřazení symbolů a jejich indexy:
Na C T
10 7 2
Výstup - hloubka iterace odečtená od indexů 10, 7 a 2:
9 6 1
```

Na příkladu výše (Výpis 3.2) je patrné, že nebude obsahovat žádnou další vnořenou iteraci, protože počet symbolů v každé skupině je roven jedné.



Obr. 3.1: Vývojový diagram algoritmu Manber & Myers

3.3.2 Zhodnocení algoritmu Manber & Myers

Tento algoritmus je první svého druhu. Dokáže vypočítat sufixové pole v lineárním prostoru. Teoreticky je časová náročnost algoritmu *O(n log n)*, protože v každé iteraci se sníží počet cyklů díky zvyšujícímu se indexu skupin. Doba běhu je tedy závislá na nejdelší shodě sufixů, která je u genetických dat vysoká, a proto pro vytvoření sufixového pole z genetických dat není tento algoritmus vhodný. Navíc v programovém prostředí R nejsou dobře odladěny cykly, které jsou používány v každé iteraci, takže je časová náročnost s každým rekurzivním voláním funkce větší.

3.4 Sufixové pole řazené pomocí algoritmu SA-IS

Název algoritmu SA-IS znamená, že vytváří sufixové pole (*Suffix Array*) pomocí indukovaného třídění (*Induced Sorting*). Indukované třídění znamená, že využíváme již seřazené prvky k tomu, abychom odvodili pořadí ostatních.

Velká část sufixového pole je vytvořena pomocí pole LMS (odvozeno z *Left-Most S*, tzn.: vybraný symbol je typu S, pokud vlevo od něj je symbol s vyšší hodnotou). Pokud si představíte jednoduchý řetězec stejných znaků, například *"AAAAA"*, je velmi snadné vytvořit pole LMS, neboť všechny symboly jsou typu L, protože všechny symboly mají stejnou hodnotu. Z nich vytvoříme sufixové pole jednoduše tím, že vypíšeme všechny sufixy od posledního znaku. Symboly typu S jsou místa v řetězci, kvůli kterým je potřeba dalších úprav, abychom dostali výsledné sufixové pole.[6][7][8]

3.4.1 Princip výpočtu

Vstupem do algoritmu SA-IS je pouze řetězec, ze kterého chceme vytvořit sufixové pole.

V prvním kroku vytvoříme ze vstupního řetězce LMS pole (výpis 3.3). To provedeme tak, že vstupní řetězec procházíme od začátku do konce, a zjišťujeme, jestli následující znak má vyšší nebo nižší hodnotu. Pokud má následující znak nižší hodnotu, je náš znak typu L, v opačném případě je typu S. Pokud je následující znak stejný, jako ten, který máme vybraný, pokračujeme dále, dokud nenarazíme na nestejný symbol. Pokud má tento nestejný symbol nižší hodnotu, jsou všechny vynechané shodné znaky typu L, v opačném případě jsou typu S. Poslední symbol je vždy typu L, protože za ním následuje prázdný znak, který má vždy nejnižší hodnotu, a tak

je on sám typu S. V závěru tohoto kroku si označíme symboly typu S, kterým nepředchází symbol typu S a ani jim nenásleduje. Takto označené symboly budeme považovat za inicializační.

Výpis 3.3: Příklad vytvoření LMS pole

```
Vstupní řetězec a jeho LMS pole:
A A G C T G T T T A G A
S S L S L S L L L S L L S
^     ^   ^       ^     ^
```

Dalším krokem je vytvoření předběžného sufixového pole, které si označíme jako SA1. V něm začneme tím, že si toto pole rozdělíme na skupiny, jejichž počet bude odpovídat velikosti vstupní abecedy (pro A, C, G, T a mezeru bude tedy skupin 5). Velikost každé skupiny bude odpovídat celkovému počtu symbolů patřících do dané skupiny. Do takto vytvořeného pole vložíme indexy inicializačních symbolů na konec skupiny, do které patří. Symbol '#' zastupuje prázdný symbol na konci vstupního řetězce.

Výpis 3.4: Příklad inicializačního SA1 pole

```
Inicializační indexy:
0                 1 1 1 1
1 2 3 4 5 6 7 8 9 0 1 2 3
^     ^   ^       ^     ^
Vytvoření SA1 pole:
 #       A     C    G       T
{13}{- - 10 1}{4}{- - 6}{- - - -}
```

Následuje postupné induktivní zaplňování SA1 pole procházením tohoto pole zleva doprava. Doplňování probíhá tak, že vezmeme index v tomto poli, zjistíme, jaký typ symbolu leží na místě o jedna menší a pokud se jedná o symbol typu L, tak jeho index doplníme na začátek skupiny, do které patří. Pokud se bude jednat o symbol typu S, pokračujeme na další pozici v poli SA1. Pokud na další pozici v poli SA1 není žádná hodnota, tak toto pole přeskakujeme a pokračujeme na další pozici.

Výpis 3.5: Příklad doplňování SA1 pole zleva doprava

```
Vytvoření SA1 pole:
 #        A       C     G        T
{13}{ - - 10 1}{4}{ - - 6}{- - - -}
 *
{13}{12 - 10 1}{4}{ - - 6}{- - - -}
     *
{13}{12 - 10 1}{4}{11 - 6}{- - - -}
          *
{13}{12 - 10 1}{4}{11 - 6}{9 - - -}
                 *
{13}{12 - 10 1}{4}{11 3 6}{9 - - -}
                           *
{13}{12 - 10 1}{4}{11 3 6}{9 5 - -}
                              *
{13}{12 - 10 1}{4}{11 3 6}{9 5 8 -}
                                 *
{13}{12 - 10 1}{4}{11 3 6}{9 5 8 7}
```

Takto vytvořené pole ještě musíme projít směrem zprava doleva, abychom zapsali i symboly typu S na správné pozice. Opět hledáme symbol na indexu o jedna menší, než je právě procházený. Tentokrát ovšem bereme pouze symboly typu S a zapisujeme je od konce skupiny, do které patří, i když přepíšeme hodnoty vložené při inicializaci.

Výpis 3.6: Příklad doplňování SA1 pole zprava doleva

```
Vytvoření SA1 pole:
 #        A       C     G        T
{13}{12 - 10 1}{4}{11 3 6}{9 5 8 7}
                                  *
{13}{12 - 10 1}{4}{11 3 6}{9 5 8 7}
                               *
{13}{12 - 10 1}{4}{11 3 6}{9 5 8 7}
                        *
{13}{12 - 10 2}{4}{11 3 6}{9 5 8 7}
                     *
{13}{12 - 10 2}{4}{11 3 6}{9 5 8 7}
              *
Výsledné pole SA1:
{13}{12 1 10 2}{4}{11 3 6}{9 5 8 7}
 *
```

Nyní jsme získali vektor hodnot, který bohužel v některých případech neodpovídá sufixovému poli. U dlouhých řetězců může dojít k tomu, že jsou některé sousední indexy ve skupině prohozeny mezi sebou. Tento problém řeší vytvoření nové abecedy ze symbolů typu S a jejich následné setřídění do nového vektoru, který je nakonec použit jako doplnění při opětovném sestavování pole SA nejdříve zleva doprava a poté zprava doleva.

Implementace opětovného řazení s použitím nové abecedy do programového prostředí R byla sice provedena, ovšem její výsledky neodpovídaly teoretickým předpokladům ani po několikerém přizpůsobení, proto bylo od této metody upuštěno. Dalším důvodem byla velká časová náročnost kvůli velkému počtu výpočetních cyklů.

3.4.2 Zhodnocení algoritmu SA-IS

Algoritmus založený na principu indukovaného třídění by byl vhodný pro genetická data díky jeho schopnosti seřazovat podobné sufixy na základě již získaných dat. Problém ovšem nastává při implementaci tohoto přístupu do programového prostředí R. Programovací jazyk R totiž nedokáže plně využít potenciál tohoto algoritmu, a to hlavně kvůli opakujícím se cyklům, jenž nejsou dobře odladěny, a tak je výpočet velmi časově náročný. Navíc se mi v mé implementaci nepovedlo dosáh-

nout absolutně správného sufixového pole a od pokusu o další řešení jsem upustil pro velkou časovou náročnost.

3.5 Vyhledávání v sufixovém poli

Stejně jako v sufixovém stromě je vyhledávání v sufixovém poli rychlé a organizované. Hlavní rozdíl je v tom, že neprocházíme větve stromu, ale vybíráme řádky tabulky. Ukážeme si to na příkladu.[9]

Tab. 3.2: Hledání řetězce CATC ve vstupní sekvenci GCATCAT#

SA	Seřazeno		C	A	T	C
8	#					
6	AT#					
3	ATCAT#					
5	CAT#		x	x	x	
2	CATCAT#		x	x	x	x
1	GCATCAT#					
7	T#					
4	TCAT#					

Z tabulky 3.2 je vidět, že již při nalezení prvního symbolu hledaného řetězce zúžíme oblast vyhledávání na pouhá dvě místa v řetězci. Když projdeme celý hledaný řetězec zjistíme, že jeho začátek se nachází na pozici 2 v původním řetězci.

Kdybychom chtěli hledat například řetězec „CAG“, tak první dvě pozice nám budou ukazovat na indexy 2 a 5, ale pro třetí pozici nám již systém vypíše, že takový řetězec ve vstupních datech neexistuje.

3.6 Nalezení shod dvou různých vstupů pomocí sufixového pole

K tomuto účelu nám dobře poslouží pole LCP zmíněné již v kapitole 3.1. Jedná se o pomocné pole hodnot, které udává počet shodných prvků po sobě jdoucích sufixů v sufixovém poli. Pokud vytvoříme sufixové pole obsahující obě porovnávané sekvence rozlišené například posledním znakem, můžeme určit shodné části mezi těmito dvěma sekvencemi.

4 CELOGENOMOVÉ ZAROVNÁNÍ

Zarovnání sekvencí je základem pro mnoho metod analýzy sekvencí. Většina algoritmů pro zarovnávání sekvencí funguje na principu postupného heuristického přibližování. Tento přístup ovšem čelí problémům s výpočetní náročností při zarovnávání velkých datových souborů. Některé metody jsou uzpůsobeny pro zpracování velkého objemu dat, ale za cenu zvýšení míry chybovosti. Jiné zase mají chybovost téměř nulovou, ale nejsou vhodné pro práci s velkými soubory kvůli jejich časové náročnosti.[10]

V této kapitole si představíme tři nejznámější metody zarovnávání a zhodnotíme jejich přednosti a nedostatky.

4.1 Needlemanův-Wunschův algoritmus

Needlemanův-Wunschův algoritmus se používá v bioinformatice a slouží k zarovnávání nukleotidových nebo proteinových sekvencí. Byl navržen Saulem B. Needlemanem a Christianem D. Wunschem, kteří ho publikovali v roce 1970.[11][12]

4.1.1 Princip algoritmu

Vstupem funkce je takzvaná skórovací matice, dvě vstupní genetické sekvence a penalizace mezery.

V prvním kroku vytvoří algoritmus matici, jejíž názvy sloupců odpovídají prvkům první vstupní sekvenci a názvy řádků prvkům sekvence druhé. Následně jsou vygenerovány hodnoty prvního řádku a prvního sloupce jako postupné přičítání penalizace mezery.

Tab. 4.1: Výchozí tabulka pro Needlemanův-Wunschův algoritmus pro vstupní nukleotidové řetězce ATACGTC a AACTTCA a penalizaci mezery -1.

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

Při zpracování porovnáváme tři směry: zleva, shora a úhlopříčně. Hodnotu z úhlopříčného směru získáme tak, že pokud se znaky shodují, přičteme k úhlopříčné hodnotě ohodnocení shody, pokud se neshodují, přičteme ohodnocení neshody. Druhou hodnotu vypočteme přičtením penalizace mezery k hodnotě shora a poslední třetí hodnotu získáme přičtením penalizace mezery k hodnotě zleva. Z těchto tří hodnot vybereme maximální, a tu zapíšeme do daného políčka spolu se šipkou určující směr, ze kterého jsme hodnotu získaly.

Tab. 4.2: Tabulka vytvořená Needleman-Wunsch algoritmem pro vstupní nukleotidové řetězce ATACGTC a AACTTCA s penalizaci mezery -1, ohodnocením shody 1 a neshody -1. Zpětná cesta vyznačena tučně.

		A	A	C	T	T	C	A
	0	-1	-2	-3	-4	-5	-6	-7
A	-1	↖ **1**	← 0	← -1	← -2	← -3	← -4	← -5
T	-2	**↑ 0**	↖ 1	← 0	↖ 0	← -1	← -2	← -3
A	-3	↑ -1	↖ **1**	↖ 1	← 0	↖ 0	← -1	↖ -1
C	-4	↑ -2	↑ 0	↖ **2**	← 1	← 0	↖ 1	← 0
G	-5	↑ -3	↑ -1	↑ 1	↖ **2**	← 1	↑ 0	↖ 1
T	-6	↑ -4	↑ -2	↑ 0	↖ 2	↖ **3**	← 2	← 1
C	-7	↑ -5	↑ -3	↑ -1	↑ 1	↑ 2	↖ **4**	**← 3**

Výpis 4.1: Výstup algoritmu Needleman-Wunsch - zarovnané sekvence.

```
A - A C T T C A
|   | |   | |
A T A C G T C -
```

4.1.2 Zhodnocení algoritmu

Algoritmus Needleman-Wunsch je vhodný především pro zarovnání kratších sekvencí o podobné délce. Za výhody tohoto algoritmu lze považovat jednoduchost a přesnost. Nevýhodou je velká výpočetní náročnost, která je *O(mn)*, kde m je délka první sekvence a n délka sekvence druhé. Algoritmus vypočítává pro každé porovnání matici o stranách, jenž jsou dlouhé jako vstupní sekvence, proto je nevhodný pro velké datové soubory a pro zarovnání více sekvencí, ale lze jej použít pro globální zarovnání a jeho přesnost závisí na zvolené skórovací matici.

4.2 Smithův-Watermanův algoritmus

Algoritmus se používá v bioinformatice pro lokální zarovnávání proteinových nebo nukleotidových sekvencí, tedy pro hledání nejpodobnějších úseků mezi dvěma sekvencemi. Tento algoritmus sestavili v roce 1981 Temple F. Smith a Michael S. Waterman.[11][13][14]

4.2.1 Princip algoritmu

První krok algoritmu vytvoří podobnou tabulku, jako algoritmus Needleman-Wunsch (viz. tabulka 4.1) s tím rozdílem, že v prvním řádku a v prvním sloupci jsou místo penalizace mezery zapsány samé nuly.

V dalším kroku počítáme stejně, jako u algoritmu Needleman-Wunsch, ale pokud nám maximální číslo vyjde záporné, zapíšeme místo něj nulu.

Zpětnou cestu získáme nalezením maximálního čísla v tabulce a postupným hledáním maxim směrem k levému hornímu rohu. Hledání ukončíme, když narazíme na nulu.[11]

Tab. 4.3: Tabulka vytvořená Smith-Waterman algoritmem pro vstupní nukleotidové řetězce AACCTATAGCT a GCGATATA s penalizaci mezery -1, ohodnocením shody 1 a neshody -1. Zpětná cesta vyznačena šipkami.[11]

		A	A	C	C	T	A	T	A	G	C	T
	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	0	1	0	0
C	0	0	0	1	1	0	0	0	0	0	2	1
G	0	0	0	0	0	0	0	0	0	1	0	1
A	0	1	1	0	0	0	1	0	1	0	0	0
T	0	0	0	0	0	↖1	0	2	1	0	0	1
A	0	1	1	0	0	0	↖2	0	3	2	1	0
T	0	0	0	0	0	1	1	↖3	2	2	1	2
A	0	1	1	0	0	0	2	2	↖4	3	2	1

Výpis 4.2: Výstup algoritmu Smith-Waterman - nejpodobnější subsekvence.[11]

```
T    A    T    A
|    |    |    |
T    A    T    A
```

4.2.2 Zhodnocení algoritmu

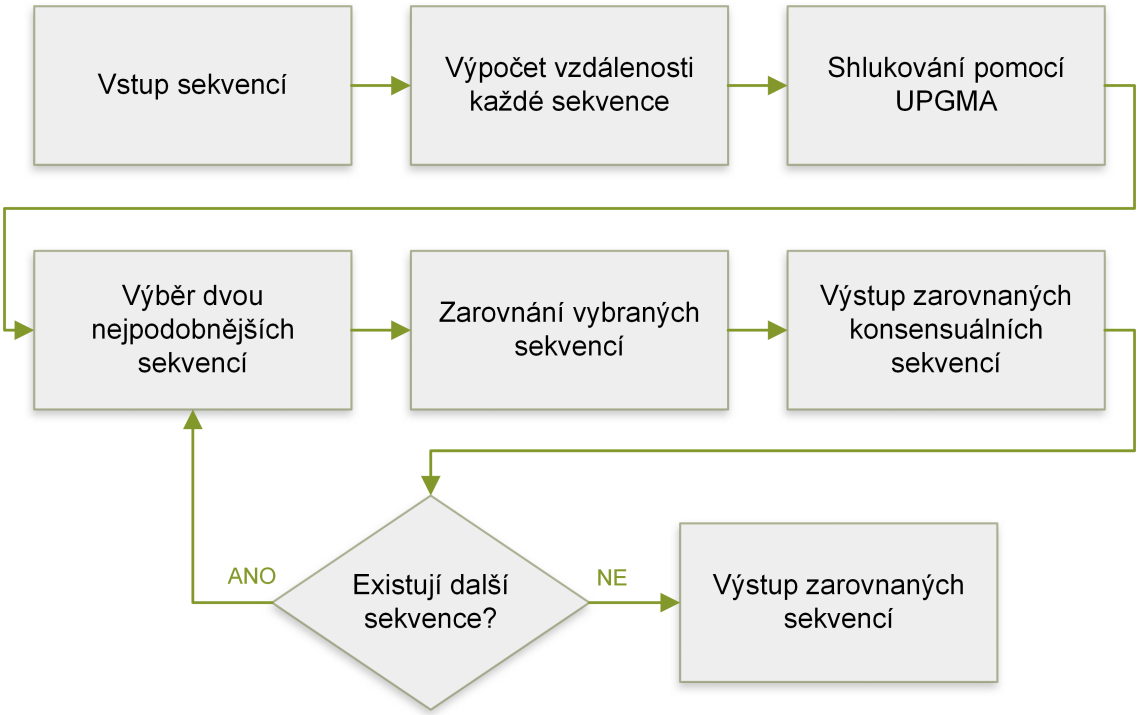
Tento algoritmus je vhodný pro lokální zarovnání kratších sekvencí, které mohou mít rozdílnou délku. Algoritmus není vhodný pro velké objemy dat z důvodu velké výpočetní a časové náročnosti.

4.3 Program Clustal Omega

Program Clustal Omega je nejnovějším členem rodiny programů pro zarovnávání s názvem Clustal. Ve srovnání s předchozími verzemi nabízí výrazné zvýšení škálovatelnosti. Pomocí něj lze zarovnat více než sto tisíc sekvencí, a to v poměrně krátkém čase, řádově v hodinách.[15][16]

Jako předzpracování využívá metodu shluků UPGMA. Jedná se o algoritmus pro tvorbu fylogenetických stromů neboli příbuzenských vztahů na základě genetické podobnosti. [11]

Samotné zarovnání využívá skrytých Markovových modelů. Tyto modely pracují s určitou pravděpodobností, a proto mohou zanášet do výsledku mírné nepřesnosti.[10][15][16][17]



Obr. 4.1: Blokové schéma principu fungování programu Clustal Omega.

4.3.1 Výstup z programu

Vstupem do programu Clustal Omega může být jakákoli sada genetických dat. Jako příklad byl vybrán gen CDKN2A, známý jako inhibitor cyklin-dependentní kinázy 2A, zodpovědný za tvorbu proteinů p16 a p14arf. Tyto proteiny se podílí na regulaci buněčného cyklu, a působí tak jako nádorové supresory. Na obrázku 4.1 můžete vidět malou část zarovnaných sekvencí.

Výpis 4.3: Výstup programu Clustal Omega - část zarovnané sekvence genu CDKN2A čtyř různých organismů.[17]

```
NC_006127.4:c78636973-78627024
NC_000070.6:c89294619-89274473
NC_000009.12:c21995043-21967752
NC_006476.4:c22506559-22479419
CAGCAGTCAGCGCTGCAGCCCCGCGCTGTGGCCCCGCAGCGGAGGCA-GCCGTGTGAGCG
G------CAGACAGACTGGCCAGGGCGGCGGCCCAGGGCCGTGTGCATGACGTGCGGGCA
G------CTGACTGGCTGGCCACGGCCGCGGCCCGGGGTCGGGTAGAGGAGGTGCGGGCG
G------CTGACTGGCTGGCCACGGCCGCGGCCCGGGGTCGGGTAGAGGAGGTGCGGGCG
       * *     * * **   ** * ***** *   **     * *  *** * **
```

Na ukázce výstupu z programu Clustal Omega vidíme čtyři zarovnané sekvence. Jak již bylo zmíněno výše, jedná se o malou část zarovnání genu CDKN2A od čtyř různých organismů, konkrétně člověka, myši, kuřete a šimpanze. Nad nimi je umístěna hlavička každého konkrétního genu, ke kterému daná sekvence patří. Pod sekvencí vidíme hvězdičky, které značí shodný nukleotid pro všechny sekvence.

4.3.2 Zhodnocení programu

Program zvládne zarovnání i velkého množství sekvencí o velkém objemu dat. Problém je v tom, že hlavní metoda zarovnávání sekvencí je založena na práci se skrytými Markovovými modely, což je metoda pracující s pravděpodobností. Proto tento program není stoprocentně přesný. Jeho přesnost je závislá na pravděpodobnostních parametrech skrytého Markovova modelu.

4.4 Celkové zhodnocení

Vzhledem k tomu, že existuje celá řada algoritmů a programů pro zarovnávání genetických dat, které se neustále vyvíjejí, není možné přesně určit, který z nich je nejlepší a který nejhorší. Každý má své klady a zápory. Vždy záleží na tom, jaká vstupní data potřebujeme zarovnat, jak kvalitní výsledek potřebujeme a jak výkonný výpočetní stroj máme k dispozici. Z výše popsaných metod lze vyvodit, že pro přesné globální zarovnání krátkých sekvencí je vhodný algoritmus Needleman-Wunsch a naopak pro globální zarovnání dlouhých sekvencí, ale s určitou mírou chybovosti, je vhodný program Clustal Omega.

5 PROGRAM

Programů pro tvorbu sufixových stromů je celá řada. Většina z nich využívá Ukkonenův algoritmus (viz. kapitola 2.1) jsou však napsány v rozdílných jazycích. Je to proto, že Ukkonenova metoda řeší tvorbu sufixového stromu v lineárním čase, což zaručuje dobré výsledky i při velmi dlouhých vstupech. První vytvořený program vychází taktéž z Ukkonenova algoritmu a je psán v jazyce R, ale k výslednému zarovnávání nakonec nebyl použit z důvodu složitosti vyhledávání ve výstupních datech a nedokonalé implementaci, kvůli které vznikaly na výstupu nepřesnosti.

Vzhledem k tomu, že sufixové pole je datová struktura mající podobné vlastnosti jako sufixový strom, a také obsahující shodné informace, byl pro výsledný program použit algoritmus využívající právě sufixová pole. Pro konstrukci sufixového pole nebyl použit žádný z algoritmů popsaných v kapitole 3. Byl použit vlastní algoritmus, který bude vysvětlen v této kapitole.

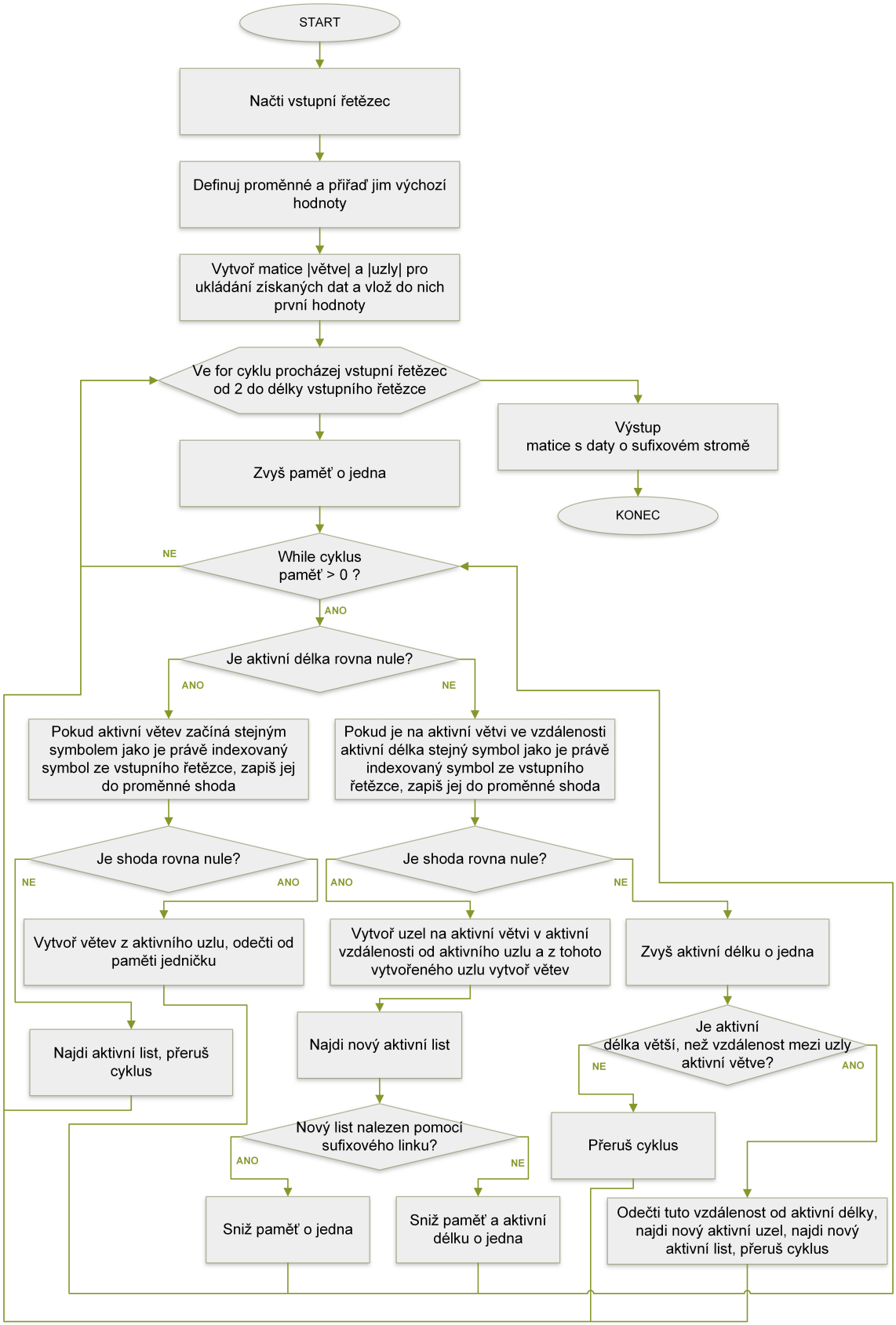
5.1 Programovací jazyk R

R je programovací jazyk primárně určený pro práci se statistickou analýzou dat a pro grafické zobrazení výsledků analýzy. Jeho předchůdcem byl komerční programovací jazyk S, ale z důvodu svobodné licence OpenSource, pod kterou je jazyk R vydáván, v dnešní době převyšuje svého předchůdce počtem uživatelů hned několikanásobně.

Tento programovací jazyk navíc disponuje jednoduchou funkcí importu knihoven. Hlavní knihovnou, využívanou v tomto projektu, je knihovna Biostrings od společnosti Bioconductor. Do knihovny Biostrings je implementováno několik užitečných funkcí pro práci s biologickými řetězci, jako je DNA, RNA a podobně.

5.2 Tvorba sufixových stromů

Vstupem programu pro tvorbu sufixových stromů je jakákoli sekvence uložená ve formátu DNAString. Program si ji ze začátku upraví podle svých potřeb a pak s ní dále pracuje. Nejprve si definuje proměnné, které bude využívat, a dvě matice. První matice bude sloužit pro uložení všech listů tvořeného stromu a druhá bude odpovídat všem vytvořeným uzlům. Matice lze vidět ve výpise 5.1 níže. Dále program prochází ve for cyklu celou sekvenci prvek po prvku. V každém cyklu se kontroluje splnění některého ze tří pravidel vysvětlených v kapitole 2.1.[3]



Obr. 5.1: Vývojový diagram zjednodušeně popisující funkci navrženého programu na tvorbu sufixového stromu.

Výpis 5.1: Výstup z programu při vstupu krátkého řetězce GCATCAT.

```
[[1]]
  zacatek konec uzelS uzelK
1       1     8     0   Inf
        2     4     0     1
        3     4     0     2
        4     4     0     3
        5     8     1   Inf
        8     8     1   Inf
        5     8     2   Inf
        8     8     2   Inf
        5     8     3   Inf
        8     8     3   Inf
        8     8     0   Inf

[[2]]
  uzel ->uzel<- propojeni
1    0        0         3
     1        0       Inf
     2        0         1
     3        0         2
```

Výstupem jsou již zmíněné dvě matice. První z nich obsahuje všechny vytvořené větve. Sloupce *zacatek* a *konec* nám prozrazují začáteční a koncový index řetězce znaků ze vstupní sekvence, *uzelS* říká, ze kterého uzlu je větev vedena a *uzelK* odpovídá uzlu koncovému. Pokud je *uzelK* roven *Inf* jedná se o poslední list dané větve.

Druhá tabulka obsahuje tři sloupce. První sloupec identifikuje jednotlivé uzly navzájem od sebe - každému uzlu je přiřazena specifická hodnota, aby byly mezi sebou rozeznatelné. Ve druhém sloupci je identifikační číslo uzlu, ze kterého daný uzel vystupuje. Třetí sloupec ukazuje na uzel, ze kterého do daného uzlu vede sufixový link.

5.2.1 Zhodnocení programu pro tvorbu sufixových stromů

Jak již bylo zmíněno výše, program není bezchybný a výstupy z něj se nedají dobře použít pro zarovnávání sekvencí. Po drobných úpravách by se dal použít pro konstrukci ukázkových sufixových stromů, které by bylo možné dále zpracovávat. Pro celogenomové zarovnání bohužel není tato implementace použitelná.

5.3 Tvorba sufixových polí

Program je vytvořen tak, aby nezáleželo na tom, jaká datová struktura z knihovny Biostring je použita. Genetický řetězec je totiž v prvním kroku převeden na numerické pole, se kterým se mnohem lépe pracuje.

Vzhledem k tomu, že chceme porovnávat dva řetězce, je nutné tyto dva řetězce spojit a při porovnávání odlišit. Z toho důvodu si před spojením uložíme délku prvního řetězce do nezávislé proměnné pojmenované například *velikost*. Tuto proměnnou využijeme až při tvorbě LCP.

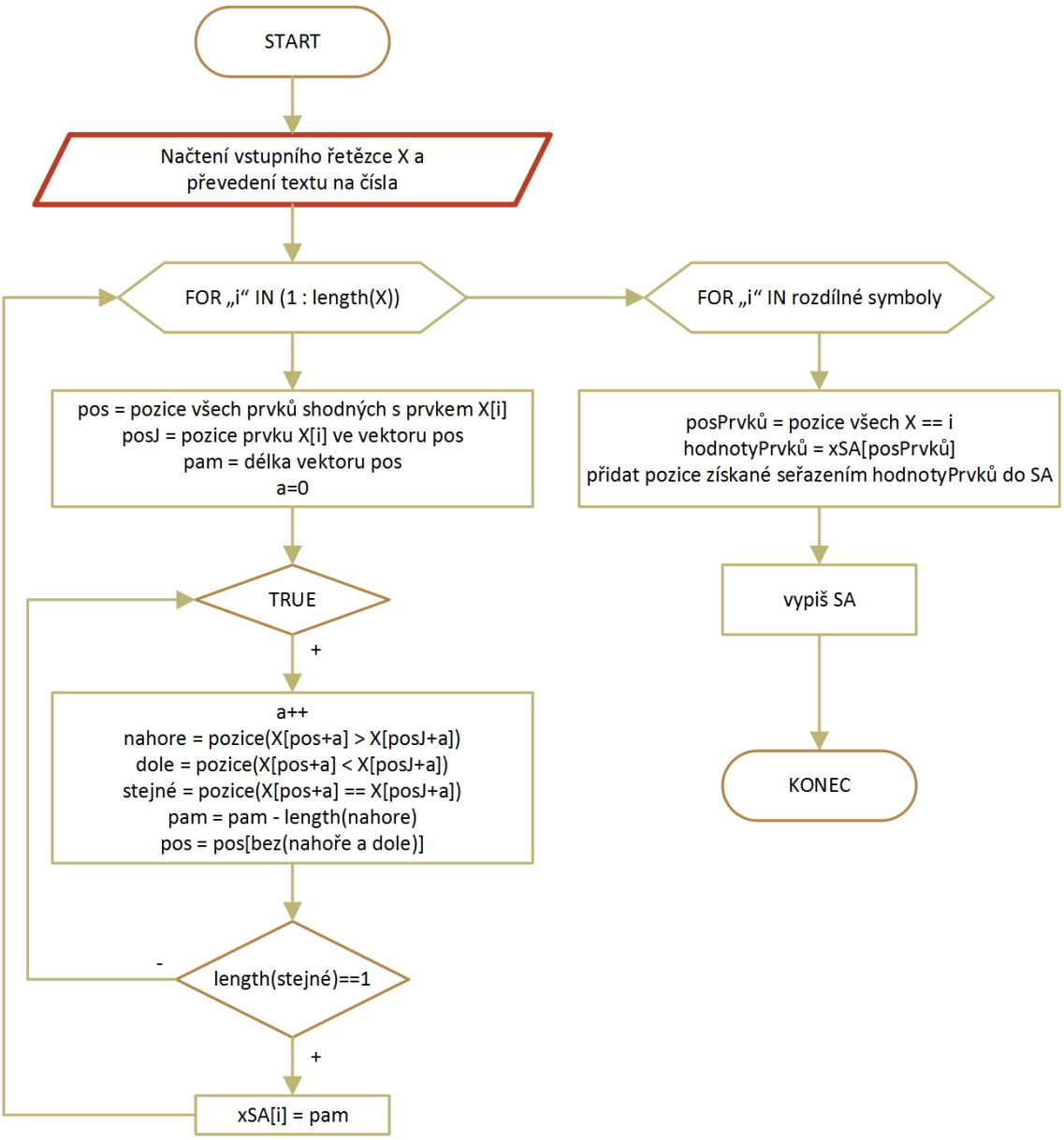
Po načtení dat následuje nejdéle trvající část algoritmu, kterou je tvorba sufixového pole. V první fázi tento algoritmus prochází prvek po prvku a pro každý zapisuje jeho pozici vůči ostatním prvkům stejné hodnoty. Tato pozice je získána pomocí třídění následujících prvků od všech prvků shodných s prvkem hledaným. Následně se zjistí, které prvky jsou menší, které shodné a které větší. Počet menších prvků je uložen jako předpokládaná hledaná pozice. Pokud se počet shodných prvků rovná jedné, je tato předpokládaná pozice i pozicí hledanou. V opačném případě se od těchto shodných prvků zjistí jejich následující, a ty se pak opět seřadí, zjistí se, které jsou menší, které shodné a které větší a proces se opakuje, dokud počet shodných prvků není roven jedné.

Výpis 5.2: Příklad tvorby sufixového pole

```
Vstupní data:
A A C T C T G G A
Vektor hodnot skupinových pozic:
2 3 1 1 2 2 2 1 1
Setřídění do skupin a doplnění indexů pozic
ze vstupního řetězce:
A A A C C G G T T
2 3 1 1 2 2 1 1 2
1 2 9 3 5 7 8 4 6
Seřazení indexů pozic vstupního řetězce podle hodnot
skupinových pozic - získání SA:
9 1 2 3 5 8 7 4 6
```

Po projití celého řetězce získáme vektor skupinových hodnot. Ten je pro názornost uveden ve výpisu 5.2, kde je i naznačeno další zpracování, které spočívá v setřídění vstupních dat do skupin podle shodných prvků a zapsání jejich pozic ve vstupním vektoru.

Posledním krokem je setřídění indexů ze vstupního vektoru podle hodnot skupinových pozic v dané skupině, čímž získáme výsledné sufixové pole.



Obr. 5.2: Vývojový diagram části programu na vytvoření sufixového pole

Po vytvoření sufixového pole potřebujeme získat inverzní sufixové pole, které vytvoříme jednoduše tak, že použijeme funkci *order*, která nám vrátí vektor indexů uspořádaných prvků, na vektor SA.

Posledním krokem před samotným zarovnáváním je nalezení společných prefixů sousedních sufixů. K tomu použijeme Kasaiův algoritmus [5][18][19]. Kasaiův algoritmus dokáže pracovat s lineární časovou náročností. Ke konstrukci LCP pomocí tohoto algoritmu potřebujeme SA, ISA a vstupní řetězec.

5.4 Celogenomové zarovnání

Samotné zarovnání se provádí pomocí pole LCP. Toto pole obsahuje shody všech sufixů, a tedy i shody ve vyhledávací sekvenci, proto při hledání shod mezi první a druhou sekvencí musíme ověřovat, zda se jedná o hodnotu LCP nalezenou mezi rozdílnými sekvencemi. Nebereme každou shodu, která existuje, ale pouze ty, které překročí předem definovaný práh. Tento práh si uživatel definuje ve vstupních parametrech funkce.

Při nalezení všech shod, které jsou nad definovaným prahem, vznikne několik duplicitních hodnot, protože se jednotlivé rozsahy společných hodnot překrývají. Dalším krokem je proto odstranění těchto duplicit, a tím zpřesnění vyhledávání hledaného shodného úseku dat.

Pro práci se všemi jednotlivými body je potřeba velké množství paměti a výpočetního výkonu. Proto v následujícím kroku hledáme pouze ty body, které určují počátek a konec přímek, znázorňujících hledané shodné úseky. Pro detekci počátečního bodu úsečky využíváme euklidovskou vzdálenost od počátku souřadného systému. Pro detekci dalších bodů je využito faktu, že úsečka pokračuje ve směru vektoru $\vec{u} = [1, 1]$

5.5 Výstupy z programu

Program je koncipován jako funkce, kterou stačí jednoduše načíst příkazem *source* v programovém prostředí R. Vstupem této funkce jsou dvě sekvence, které chceme porovnávat. Je důležité, aby byli sekvence ve struktuře DNAString nebo AAString.

Dalšími vstupními parametry funkce jsou *minimální vzdálenost* a *maximální mezera*. Tyto dvě hodnoty jsou důležité pro následnou detekci začátků a konců shodných sekvencí.

Minimální shoda nám udává, kolik shodných nukleotidů nebo aminokyselin musí mezi sebou obě sekvence mít, abychom tuto část považovali za podstatnou. Metodou pokus-omyl bylo zjištěno, že ideální hodnotou minimální shody pro nukleotidy je 20 a pro aminokyseliny 15.

Maximální mezera je volitelný parametr, který je ve výchozím nastavení shodný s parametrem minimální shoda. Maximální mezera určuje, jak velká mezera může být mezi shodami, abychom sekvence stále považovali za podobné. Díky tomu se omezí vliv mutací.

Hlavním výstupem z programu je graf, který názorně ukazuje shodné části sekvencí. Dalším výstupem jsou dva vektory čísel. Ty odpovídají souřadnicím úseček shodných částí genomů.

5.5.1 Hledání mutovaného genu v originální sadě dat

V této kapitole si ukážeme, tak program zvládá hledat podobné sekvence, které se liší určitým procentem mutace. K analýze budeme používat náhodný výběr jedenácti genů z bakteriálního genomu druhu Treponema pallidum linie SamoaD. Tento náhodný výběr spojíme do jedné sekvence a použijeme ji jako referenční. Jeden z oněch jedenácti náhodně vybraných genů si uložíme i do nové proměnné. Na tomto genu provedeme mutaci v 1%, 2%, 5% a 10% případů. Následně zjišťujeme, jaké nastavení je potřeba použit, aby byl gen nalezen i s daným procentem mutace.

Tab. 5.1: Nalezené nejdelší shody pro různé procento mutace

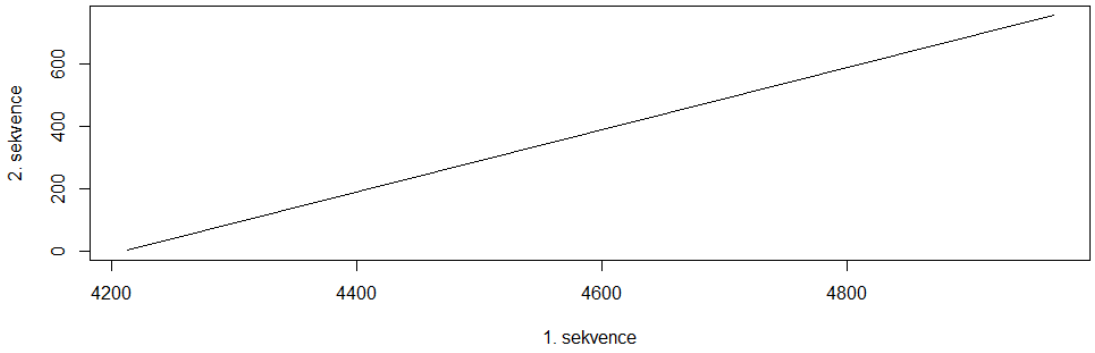
	Min. shoda: 20 Max. mezera: 20	Min. shoda: 10 Max. mezera: 20	Min. shoda: 5 Max. mezera: 20	Min. shoda: 5 Max. mezera: 30
1%	4213-4968			
2%	4213-4841	4213-4968		
5%	4213-4462	4213-4968		
10%	4629-4689	4213-4415	4213-4699	4213-4968

Správná pozice hledaného genu je 4213-4968.

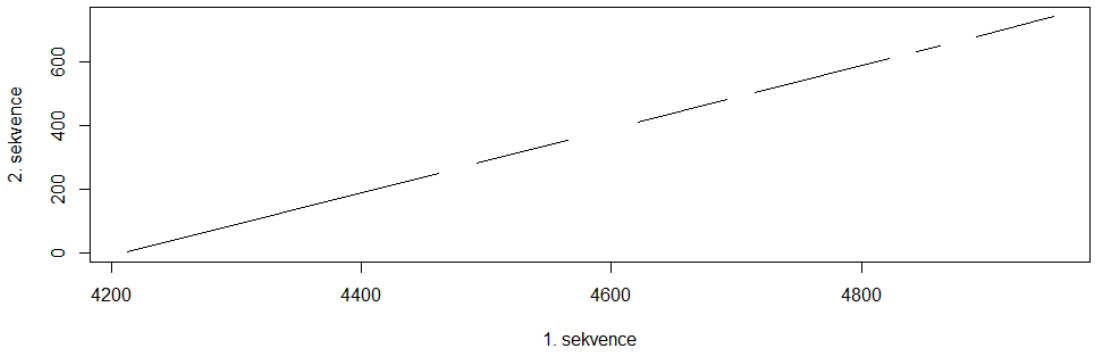
V tabulce 5.1 je vidět, že pro 1% mutace není potřeba nijak nastavovat vstupní funkci. Stačí zvolit minimální shodu i maximální mezeru na doporučenou hodnotu 20.

V případě 2% a 5% mutace již bylo nutné minimální shodu zmenšit, abychom nalezli hledanou sekvenci a úspěšně eliminovali vliv mutace.

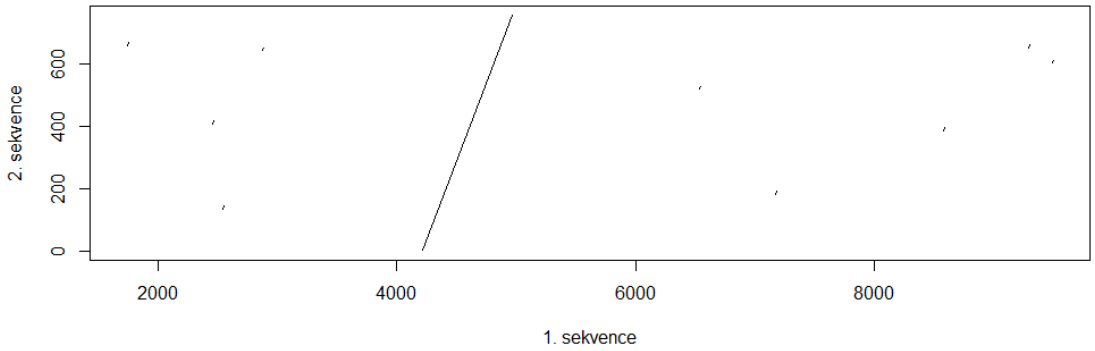
Pro 10% mutaci už nestačilo ani další snížení minimální shody, a tak bylo nutné zvýšit hodnotu maximální mezery, čímž se nám opět podařilo docílit eliminace mutací.



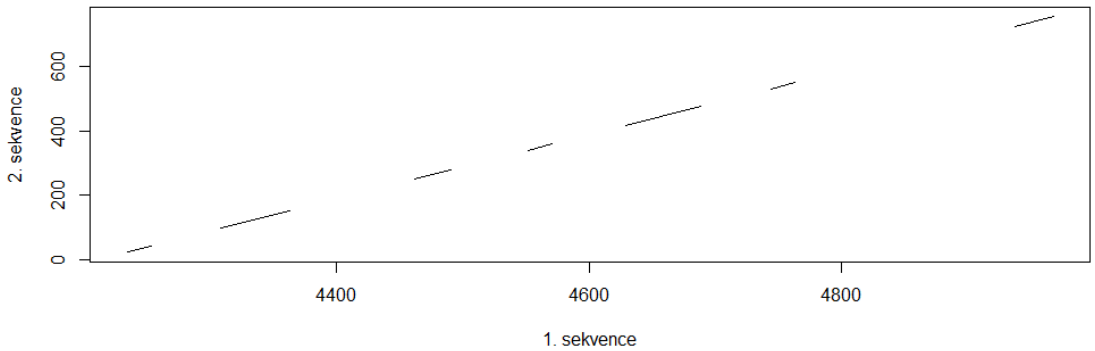
Obr. 5.3: Graf ukazující výstup z programu pro 1% mutace při minimální shodě 20 a maximální mezeře 20



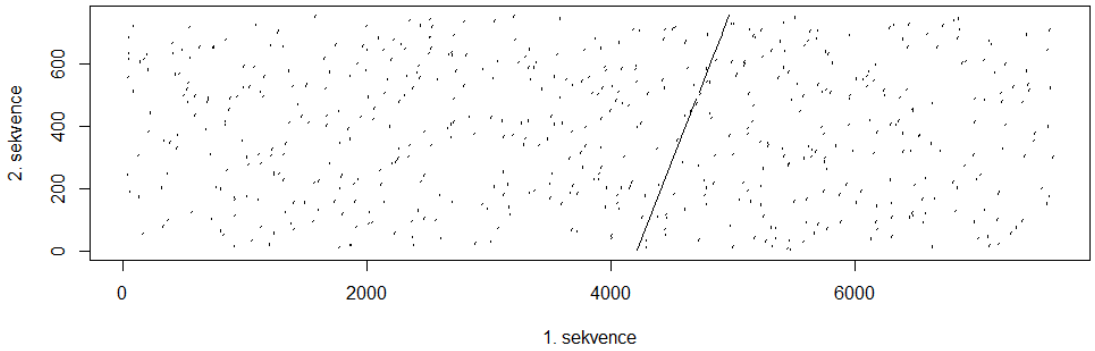
Obr. 5.4: Graf ukazující výstup z programu pro 5% mutace při minimální shodě 20 a maximální mezeře 20



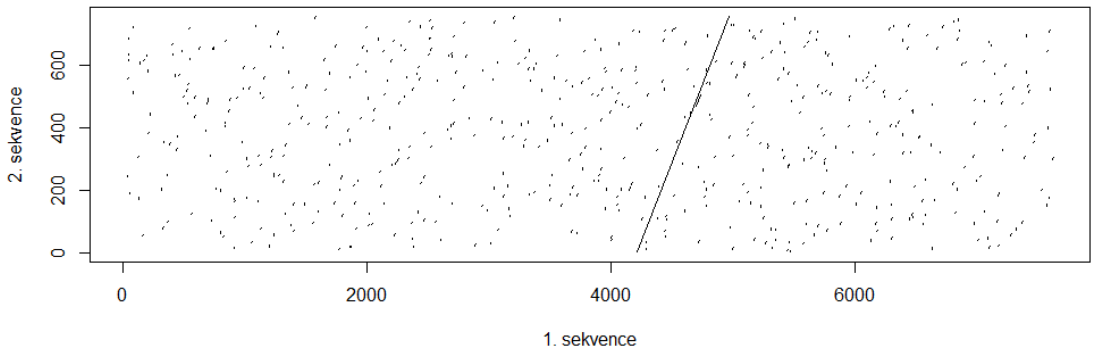
Obr. 5.5: Graf ukazující výstup z programu pro 5% mutace při minimální shodě 10 a maximální mezeře 20



Obr. 5.6: Graf ukazující výstup z programu pro 10% mutace při minimální shodě 20 a maximální mezeře 20



Obr. 5.7: Graf ukazující výstup z programu pro 10% mutace při minimální shodě 5 a maximální mezeře 20



Obr. 5.8: Graf ukazující výstup z programu pro 10% mutace při minimální shodě 5 a maximální mezeře 30

Na grafu 5.7 můžeme vidět, že i když máme nastavenou minimální shodu na velikost pouhých 5 shod, je zde stále mezera, která je pravděpodobně způsobena mutacemi, jež jsou velmi blízko sebe. Při zvětšení tolerance velikosti mezery z 20 na 30 (5.8) můžeme vidět, že se mezera zaplnila, a my tak získali celou podobnou sekvenci.

Je však důležité zmínit, že pokud mutace zasáhne začátek nebo konec hledané sekvence, tento algoritmus není schopen získat přesnou inicializační pozici a shodu vyhodnotí až na následujícím prvku. První mutovaný prvek bude ze shody vyloučen.

V grafech 5.5, 5.7 a 5.8 je patrné, že byly detekovány i falešné shody, neboli shodné úseky, které jsou již na první pohled umístěné mimo hlavní diagonálu. To je způsobeno právě snižováním minimální shody a tím zvyšování tolerance k falešným detekcím.

5.5.2 Hledání referenčního genu v celém genomu

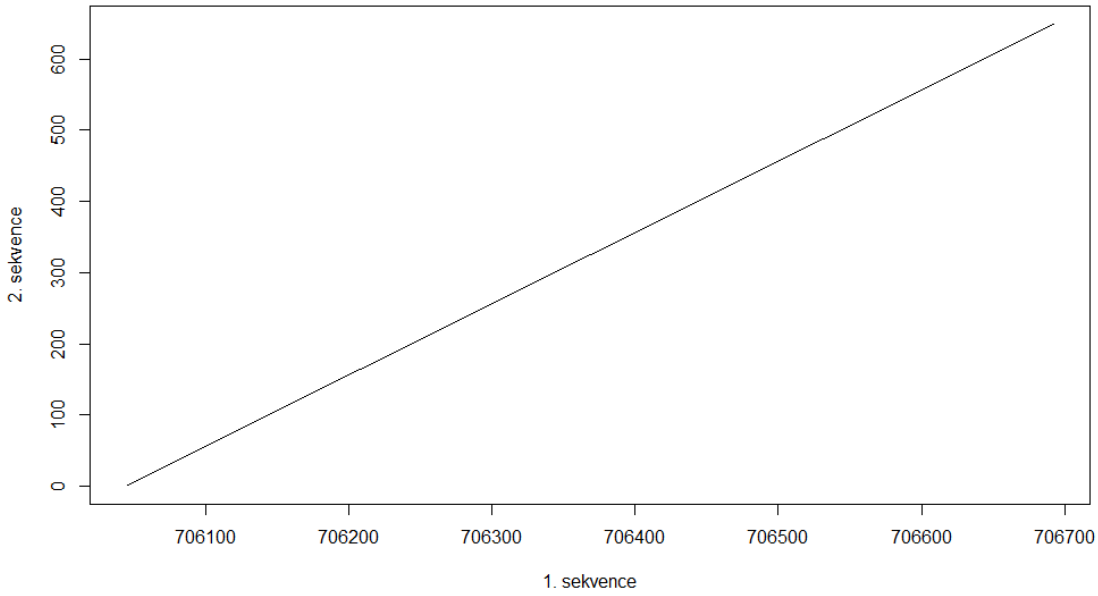
K testování byly použity bakteriální genomy druhu Treponema pallidum linií SamoaD a následně byl hledán v liniích Chicago, CDC2, MexicoA a FribourgB.

Jako referenční gen byl vybrán gen dnaQ, jehož produktem je DNA polymeráza III podjednotka epsilon. Tento produkt slouží jako editační složka při replikaci bakterií [20].

Tab. 5.2: Výsledky z celogenomového zarovnání referenčního genu

Organismus	**Nalezená pozice genu**	**Skutečná pozice genu**	**Doba výpočtu [s]**	**Délka genomu [pb]**
Chicago	706045-706692	706045-706692	87968	1139281
CDC2	706840-707487	706840-707487	84614	1139744
MexicoA	706776-707423	706776-707423	83697	1140038
FribourgB	707106-707753	707106-707753	84533	1140481

V tabulce 5.2 je vidět, že pokud je gen v genomu shodný s genem referenčním, dostaneme stoprocentní shodu v pozici hledaného genu. Doba výpočtu se pohybuje kolem 23 - 25 hodin. Tato doba závisí jednak na výpočetním výkonu stroje, na kterém program běží a také na délce shodných částí, které se v genomu vyskytují.



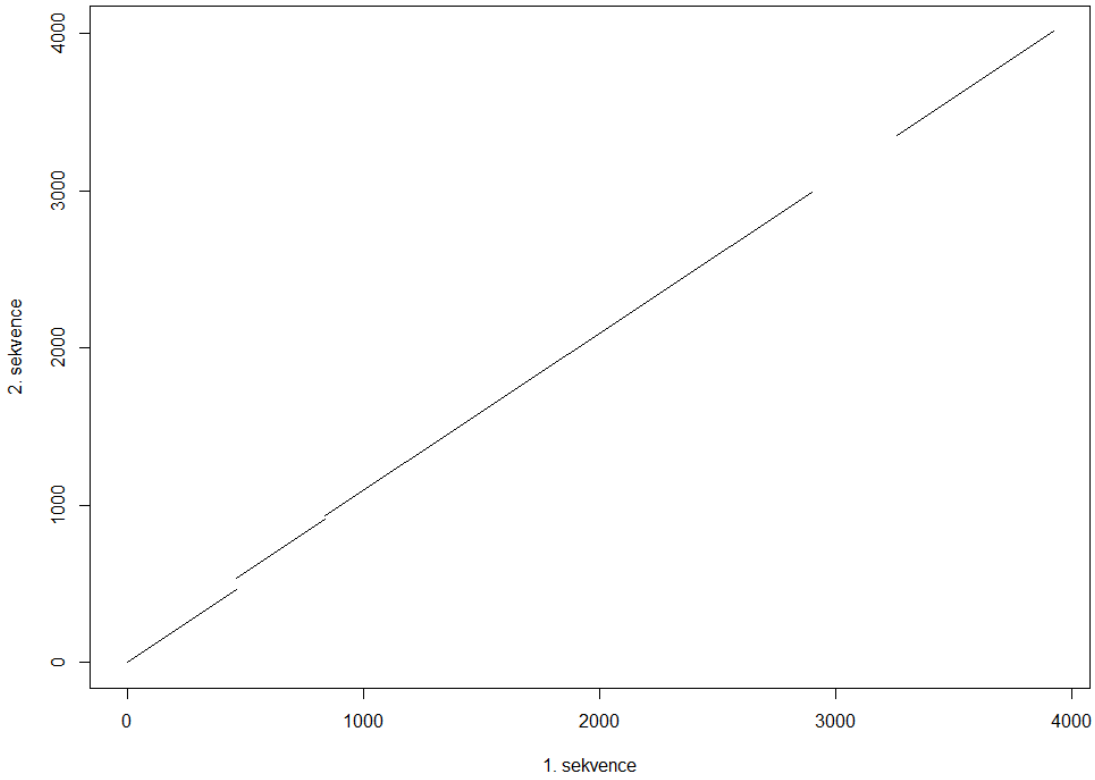
Obr. 5.9: Graf ukazující výstup z programu pro zarovnání sekvence genu dnaQ z genomu SamoaD s celým genomem Chicago

Graf 5.9 znázorňuje grafický výstup z programu, který na ose X (1. sekvence) ukazuje, kde leží referenční gen (2. sekvence). Přesné hodnoty (tabulka 5.2) jsou následně vypsány do konzole.

Pro výpočet bylo použito výpočetní zařízení s operační paměť 24 GB a s procesorem Intel® Xeon® X5690 pracujícím na taktu 3,47 GHz.

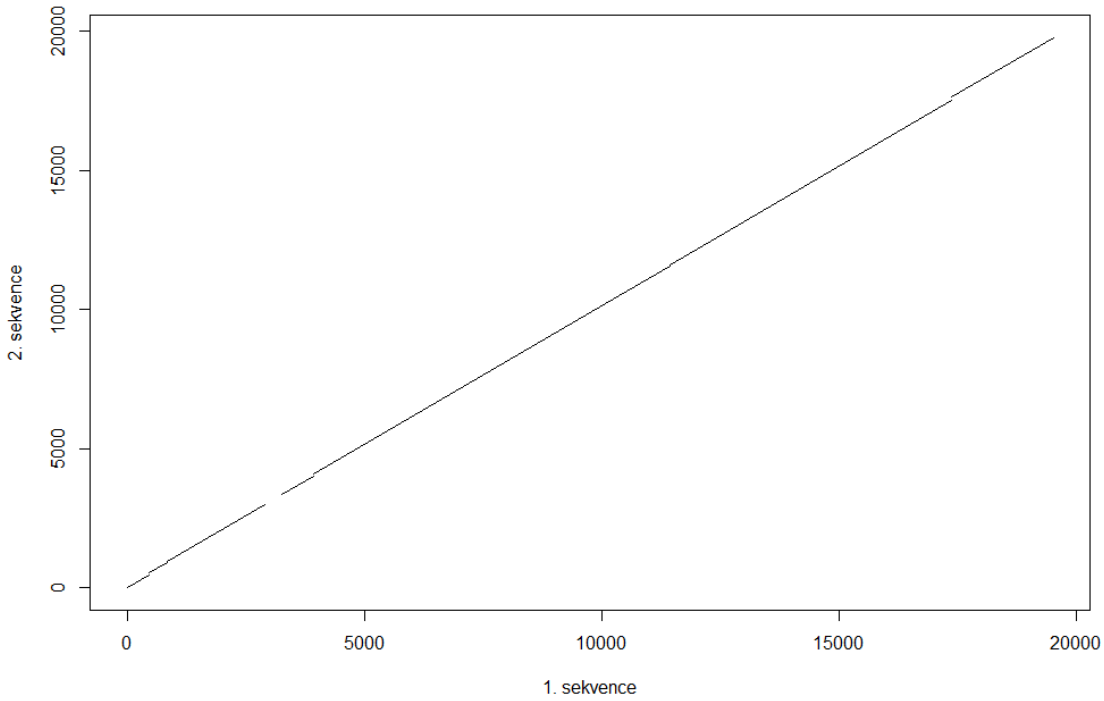
5.5.3 Zarovnání dvou genomů

Zarovnat dva celé genomy by naším programem trvalo asi 200 dní. Proto si v této kapitole ukážeme zarovnání pouze úseků stejně dlouhých částí genomů převedených na aminokyseliny. Zarovnání bylo počítáno na osobním počítači s procesorem Intel Core2 Quad Q9400 2,66 GHz a s operační pamětí 4 GB RAM.



Obr. 5.10: Graf ukazující výstup z programu pro zarovnání části genomu SamoaD [1:10] se stejně velkou částí genomu Chicago

Na grafu 5.10 můžeme vidět dvě zarovnané sekvence. První sekvence je vytvořena z prvních deseti genů genomu Chicago a druhá sekvence je vytvořena z prvních deseti genů genomu SamoaD. Je patrné, že tyto vybrané sekvence nejsou úplně totožné, ale mají čtyři podobné části. Výpočetní čas tohoto zarovnání byl 330 vteřin.



Obr. 5.11: Graf ukazující výstup z programu pro zarovnání části genomu SamoaD [1:50] se stejně velkou částí genomu Chicago

Graf 5.11 je shodným zarovnáním jako graf 5.10, jen je vytvořen z většího počtu genů, konkrétně z prvních 50. Z tohoto grafu je parná podobnost obou genů, ale z důvodu velkého rozsahu již nejsou viditelné detaily. Tyto detaily můžeme vyčíst z výstupních vektorů (tabulka 5.3), jenž nám prozrazují, které části genomu jsou si podobné a které nikoli.

Například je dobře vidět, že z počátku jsou genomy shodné až do pozice 465. Po této pozici následuje v genomu SamoaD úsek, který v genomu Chicago chybí, protože další shodná pozice začíná v genomu Chicago na stejné pozici, kde v předchozím úseku skončila, kdežto v genomu SamoaD začíná až na pozici 541. Z toho lze usoudit, že genom SamoaD obsahuje úsek o délce 76 aminokyselin, který není v genomu Chicago obsažen.

Tab. 5.3: Výsledky ze zarovnání části genomů

	Souřadnice jednotlivých přímek v grafu zarovnání dvou sekvencí			
Chicago	1-465	465-836	837-2898	3260-3920
SamoaD	1-465	541-912	933-2994	3355-4015
Chicago	3920-10523	10527-11429	11429-17361	17361-19514
SamoaD	4073-10676	10678-11580	11609-17541	17630-19783

Výpočetní čas tohoto zarovnání byl 15577 vteřin.

5.6 Porovnání s jinými zarovnávacími algoritmy

V této kapitole porovnáme výsledky získané naším algoritmem s výsledky jiných dostupných aplikací. Začneme aplikacemi dostupnými na webových stránkách společnosti EMBL-EBI, a poté vyzkoušíme program MUMmer.

5.6.1 On-line zarovnávací nástroje

MSA (Multiple Sequence Alignment, zarovnání více sekvencí) je obecně zarovnávání více biologických sekvencí složené z proteinů nebo nukleových kyselin. Z výstupu těchto algoritmů lze odvodit homologii a evoluční vztahy mezi sledovanými sekvencemi.[17]

Clustal Omega je nástroj pro MSA, který využívá k vytváření zarovnání skrytých Markovových modelů a je vhodný pro středně dlouhé sekvence.

Kalign je velmi rychlý algoritmus pro MSA, který se zaměřuje na místní oblasti a je vhodný pro zarovnání velkých sekvencí.

MView transformujte výsledek vyhledávání podobnosti sekvence do zarovnaných sekvencí.

Při pokusu nechat zarovnat relativně krátký gen dnaQ s dlouhým genomem skončí tyto programy s chybovým hlášením(viz. výpis 5.3), které říká, že nebylo nalezeno žádné zarovnání.

Výjimkou byl jedině Kalign, který našel 100% shodu během několika vteřin (viz. výpis 5.4).

Výpis 5.3: Výstup vyhledávání genu v genomu programem Clustal Omega

```
ERROR: Cannot open input file. No alignment!
```

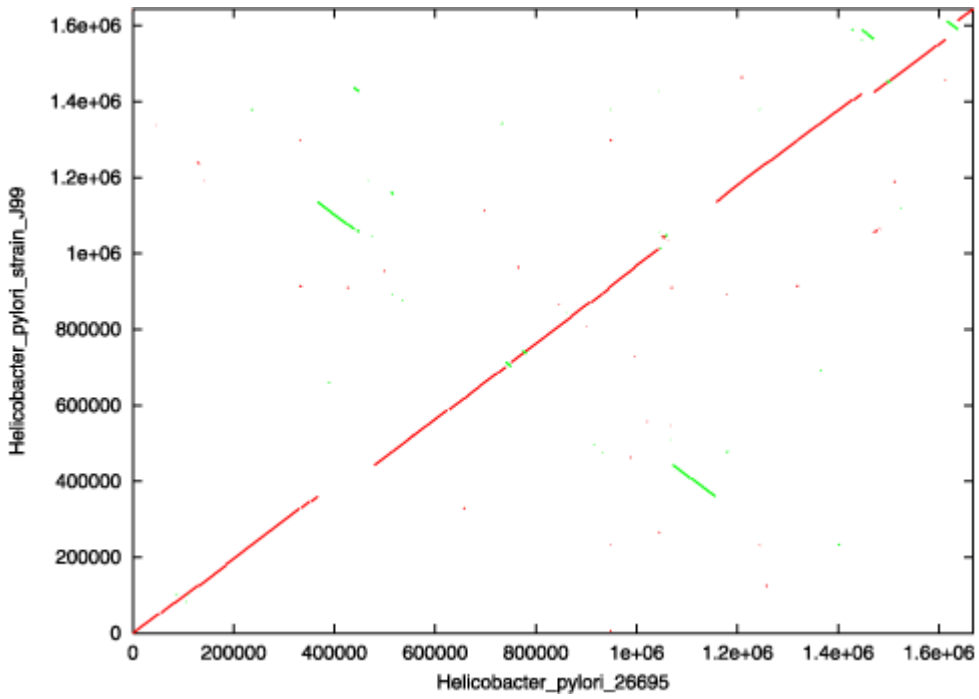
Výpis 5.4: Výstup vyhledávání genu v genomu programem Kalign

```
1: taxon          100.00  100.00
2: TPESAMD_0643   100.00  100.00
```

Pro zarovnání genů či genomů stejné nebo podobné délky jsou všechny uvedené algoritmy použitelné. Dokonce dosahují i dobrých výsledků (viz. kapitola 4.3). Jejich časovou náročnost nedokáži změřit, protože se jedná o webové aplikace bez možnosti spuštění na jiné platformě. Z toho důvodu ani nelze přesně určit, jaký počítač se stará o výpočet těchto algoritmů.

5.6.2 Program MUMmer

MUMmer je softwarový balík s otevřeným zdrojovým kódem pro rychlé zarovnání velmi dlouhých sekvencí DNA a aminokyselin. Začal vznikat již v roce 1999 a stále se vyvíjí. Nejnovější verze 3.0 obsahuje i algoritmus využívající sufixového stromu, který dále zlepšuje efektivitu celého softwaru. Díky této implementaci dokáže program MUMmer nalézt přesné shody mezi dvěma sekvencemi o velikosti 5 Mpb během pár desítek sekund, za použití 90 MB paměti RAM a jednoho jádra procesoru pracujícím na taktu 1,7 GHz. Výsledky lze poté dále zpracovávat pomocí dalších implementovaných balíčků nebo je možnost si je vykreslit ve formě grafu.[21]



Obr. 5.12: Ukázkový graf znázorňující výstup z programu MUMmer jako dot-plot

Program MUMmer disponuje obrovským komplexem nástrojů pro analýzu genetických sekvencí. S naším programem je srovnatelná funkce *mummerplot*, která vykreslí graf, zobrazující podobné úseky dvou sad genetických dat. Hlavním rozdílem je rychlost zpracování, která je u programu MUMmer o několik řádů lepší, než u našeho programu.

Při porovnávání dvou genomů SamoaD a Chicago byla výsledná matice získána za méně, než 10 sekund. Výstup z programu má podobu tří sloupečků, z nichž první dva jsou počáteční indexy shody v první a v druhém genomu. Třetí sloupeček udává délku shody (viz. výpis 5.5).

Výpis 5.5: Ukázka výstupu zarovnání dvou sekvencí programem MUMmer

```
   1        1      2825
2827     2827      3615
6443     6443       211
6655     6655       397
 ...      ...       ...
```

Při pokusu získat grafická data z tohoto programu jsem se setkal s neúspěchem v podobě chyby, kterou se mi nepodařilo vyřešit ani po pročtení pomocného fóra ur-

čeného právě k těmto účelům. Důvodem nejspíš bylo využívání linuxového prostředí na virtuálním počítači.

5.6.3 Zhodnocení

Na rychlé zarovnání genetických dat je jednoznačně nejlepší program MUMmer, který předčí všechny ostatní metody rychlostí i malou paměťovou náročností.

Pokud chceme zarovnaná data dále zpracovávat, je nejlepší volbou použít zarovnávacích nástrojů na webových stránkách společnosti EMBL-EBI. Získáme tak soubor s kompletním zarovnáním našich dat. Navíc můžeme zarovnat současně i více, než jen dvě sekvence.

Náš program se z důvodu časové náročnosti jeví jako méně vhodná volba, ale na druhou stranu zvládá vyhledat podobné sekvence velmi přesně a poradí si i s genovými mutacemi. Navíc je implementován do programového prostředí R, které je volně přístupné pod svobodnou licencí a je určené pro počítačové operační systémy Windows, Mac OS i Linux.

6 ZÁVĚR

V této práci jsme se seznámili se sufixovými strukturami a zarovnáváním genetických dat. Vysvětlili jsme si tvorbu sufixových stromů a sufixových polí. Zjistili jsme, že sufixová pole jsou přehlednější a prostorově méně náročnější datové struktury, než sufixový strom. Z toho důvodu jsme přistoupili k finálnímu řešení, které je založené na sufixovém poli a nikoli na sufixovém stromu, a proto jsme popsali i několik metod pro tvorbu sufixového pole a zhodnotili jsme jejich klady a zápory pro naši implementaci.

Během práce na vytváření tohoto programu jsem narazil na několik komplikací, které se týkaly především implementace algoritmů do programového prostředí R. Toto prostředí primárně slouží k statistické analýze a při vytváření programu se ukázalo, že není vhodné pro složitější algoritmy využívající cyklící se procesy a nezvládá pracovat se složitějšími strukturami, než jsou jednoduché tabulky. Proto nebyl prvotně navržený program konstrukce sufixových stromů schopen dosáhnout požadovaných výsledků, a tak jsem se po domluvě s vedoucí práce rozhodl program předělat. V novém programu jsem pro zpracování dat využil místo sufixových stromů sufixových polí.

Implementovaný algoritmus v programovém prostředí R je založen na vlastní funkci třídění, která prochází celý řetězec prvek po prvku a určuje jeho polohu v sufixovém poli. Tato implementace není zdaleka tak účinná jako její konkurenční nástroje. To je způsobeno především nevhodným programovým prostředím, které nebylo primárně určeno k takto velkým výpočetním operacím.

Konkrétním konkurenčním programem je MUMmer, který začal vznikat již v roce 1999 a v dnešní době se jedná o velkou aplikaci obsahující mnoho užitečných funkcí. Je postaven na programovacím jazyce C a dosahuje o mnoho lepších výsledků, než jakákoli jiná dokumentovaná implementace řešení celogenomového zarovnání.

Při pokusu o implementaci již existujících algoritmů pro konstrukci sufixového pole jsem opět narazil na problém ze strany programového prostředí R týkající se velkého množství cyklů, jež vedl k velké časové náročnosti výpočtu. Z toho důvodu jsem se rozhodl vytvořit vlastní algoritmus, který by v jiném programovém prostředí nebyl efektivní, ale v prostředí programovacího jazyku R se ukázal jako časově nejméně náročný. I přes to, že se jedná o nejrychlejší z vyzkoušených algoritmů, je jeho časová náročnost velká.

Algoritmus zvládá jak zarovnání celých genomů, tak vyhledání genu v genomu. Při zarovnávání si při vhodném nastavení dokáže poradit i s mutacemi.

Výsledný program se dá načíst jako funkce, takže je velmi jednoduchý na používání i nastavení. Z důvodu velké časové náročnosti se jeví jako vhodný převážně ke studijním účelům. Věřím, že takto zpracovaný program najde uplatnění ve výuce bioinformatiky.

LITERATURA

[1] BARTON, Carl, Alice HELIOU, Laurent MOUCHARD a Solon P PISSIS. Linear-time computation of minimal absent words using suffix array. *BMC Bioinformatics*. 2014, **15**(1), -. DOI: 10.1186/s12859-014-0388-9. ISSN 1471-2105. Dostupné také z: http://bmcbioinformatics.biomedcentral.com/articles/10.1186/s12859-014-0388-9

[2] MANBER, Udi a Gene MYERS. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*. 1993, **22**(5), 935-948. DOI: 10.1137/0222058. ISSN 0097-5397. Dostupné také z: http://epubs.siam.org/doi/abs/10.1137/0222058

[3] UKKONEN, E. On-line construction of suffix trees. *Algorithmica*. 1995, **14**(3), 249-260. DOI: 10.1007/BF01206331. ISSN 0178-4617. Dostupné také z: http://link.springer.com/10.1007/BF01206331

[4] Suffix arrays: a new method for on-line string searches. *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*. Philadelphia: Society for Industrial and Applied Mathematics, 1990, s. 319-327. ISBN 0898712513.

[5] ABOUELHODA, Mohamed Ibrahim, Stefan KURTZ a Enno OHLEBUSCH. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*. 2004, **2**(1), 53-86. DOI: 10.1016/S1570-8667(03)00065-0. ISSN 15708667. Dostupné také z: http://linkinghub.elsevier.com/retrieve/pii/S1570866703000650

[6] GE NONG, SEN ZHANG a WAI HONG CHAN. Two Efficient Algorithms for Linear Time Suffix Array Construction. *IEEE Transactions on Computers*. 2011, **60**(10), 1471-1484. DOI: 10.1109/TC.2010.188. ISSN 0018-9340. Dostupné také z: http://ieeexplore.ieee.org/document/5582081/

[7] KO, Pang a Srinivas ALURU. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*. 2005, **3**(2-4), 143-156. DOI: 10.1016/j.jda.2004.08.002. ISSN 15708667. Dostupné také z: http://linkinghub.elsevier.com/retrieve/pii/S1570866704000498

[8] NONG, Ge, Sen ZHANG a Wai Hong CHAN. Linear Suffix Array Construction by Almost Pure Induced-Sorting. *2009 Data Compression Conference*. IEEE,

2009, **2009**(16), 193-202. DOI: 10.1109/DCC.2009.42. ISBN 978-0-7695-3592-0. Dostupné také z: http://ieeexplore.ieee.org/document/4976463/

[9] KASAHARA, Masahiro. a Shinichi MORISHITA. *Large-scale genome sequence processing*. Hackensack, NJ: Distributed by World Scientific, c2006. ISBN 978-1-86094-635-6.

[10] SIEVERS, F., A. WILM, D. DINEEN, et al. *Fast, scalable generation of high-quality protein multiple sequence alignments using Clustal Omega*. DOI: 10.1038/msb.2011.75. ISBN 10.1038/msb.2011.75. Dostupné také z: http://msb.embopress.org/cgi/doi/10.1038/msb.2011.75

[11] KRANE, Dan E. a Michael L. RAYMER. *Fundamental concepts of bioinformatics*. [1st ed.]. San Francisco: Benjamin Cummings, c2003. ISBN 08-053-4633-3.

[12] NEEDLEMAN, Saul B. a Christian D. WUNSCH. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*. 1970, **48**(3), 443-453. DOI: 10.1016/0022-2836(70)90057-4. ISSN 00222836. Dostupné také z: http://linkinghub.elsevier.com/retrieve/pii/0022283670900574

[13] SMITH, T.F. a M.S. WATERMAN. Identification of common molecular subsequences. *Journal of Molecular Biology*. 1981, **147**(1), 195-197. DOI: 10.1016/0022-2836(81)90087-5. ISSN 00222836. Dostupné také z: http://linkinghub.elsevier.com/retrieve/pii/0022283681900875

[14] SMITH, Temple F a Michael S WATERMAN. Comparison of biosequences. *Advances in Applied Mathematics* [online]. 1981, **2**(4), 482-489 [cit. 2017-05-17]. DOI: 10.1016/0196-8858(81)90046-4. ISSN 01968858. Dostupné z: http://linkinghub.elsevier.com/retrieve/pii/0196885881900464

[15] PAZOS, Florencio a Mónica CHAGOYEN. *Practical Protein Bioinformatics*. Springer International Publishing, 2014. ISBN 978-3-319-12726-2.

[16] SIEVERS, F., A. WILM, D. DINEEN, et al. Fast, scalable generation of high-quality protein multiple sequence alignments using Clustal Omega. *Molecular Systems Biology*. 2011, **7**(1), 539-539. DOI: 10.1038/msb.2011.75. ISSN 1744-4292. Dostupné také z: http://msb.embopress.org/cgi/doi/10.1038/msb.2011.75

[17] Clustal Omega: Multiple Sequence Alignment. *EMBL-EBI* [online]. [cit. 2017-01-02]. Dostupné z: http://www.ebi.ac.uk/Tools/msa/clustalo/

[18] FISCHER, Johannes. *Inducing the LCP-Array.* , 374. DOI: 10.1007/978-3-642-22300-6_32. Dostupné také z: http://link.springer.com/10.1007/978-3-642-22300-6_32

[19] KASAI, Toru, Gunho LEE, Hiroki ARIMURA, Setsuo ARIKAWA a Kunsoo PARK. *Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications.* , 181. DOI: 10.1007/3-540-48194-X_17. Dostupné také z: http://link.springer.com/10.1007/3-540-48194-X_17

[20] JONCZYK, P., I. FIJALKOWSKA a Z. CIESLA. Overproduction of the epsilon subunit of DNA polymerase III counteracts the SOS mutagenic response of Escherichia coli. *Proceedings of the National Academy of Sciences of the United States of America.* 1988, **85**(23), 9124-9127.

[21] DELCHER, A. L., S. KASIF, R. D. FLEISCHMANN, J. PETERSON, O. WHITE a S. L. SALZBERG. Alignment of whole genomes. *Nucleic Acids Research* [online]. 1999, **27**(11), 2369-2376 [cit. 2017-05-17]. DOI: 10.1093/nar/27.11.2369. ISSN 03051048. Dostupné z: https://academic.oup.com/nar/article-lookup/doi/10.1093/nar/27.11.2369

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

O(n)	Rychlost algoritmu v závislosti na vstupní proměnné n
pb	páry bazí, velikost genetické sekvence
SA	suffix array, sufixová matice
ISA	inverse suffix array, inverzní sufixová matice
LCP	longest common prefix, nejdelší společná předpona
SA-SI	Suffix Array - Induced Sorting, algoritmus tvorby sufixového pole na principu indukovaného třídění
UPGMA	Unweighted Pair Group Method with Arithmetic Mean, shluková analýza
MSA	Multiple Sequence Alignment, zarovnání více sekvencí

SEZNAM PŘÍLOH

1. Zdrojový kód v souboru GenomeAligment.R