Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Dátové štruktúry pre uchovávanie sekvenovacích dát

Diplomová práca

2015 **Bc. Jakub JURSA**

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Dátové štruktúry pre uchovávanie sekvenovacích dát

Diplomová práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	Mgr. Tomáš Vinář, PhD.
Konzultant:	Mgr. Vladimír Boža

Bratislava 2015 **Bc. Jakub JURSA**

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Jakub Jursa
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Dátové štruktúry pre uchovávanie sekvenovacích dát
Data Structrures for Storing DNA Sequencing Data

Cieľ: Výstupom zo sekvenovania genómu je množina veľkého množstva krátkych reťazcov. Tieto reťazce obvykle pochádzajú z jedného spoločného nadslova. Cieľom práce je navrhnúť, implementovať a otestovať niekoľko prístupov ako tieto reťazce uchovávať a indexovať pre následne vyhľadávanie so zameraním na čo najmenšiu pamäťovú náročnosť (pri zachovaní dobrej časovej zložitosti vyhľadávania).

Vedúci: Mgr. Tomáš Vinař, PhD.
Konzultant: Mgr. Vladimír Boža
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, PhD.

Dátum zadania: 24.04.2014

Dátum schválenia: 06.05.2014 prof. RNDr. Branislav Rovan, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie

Ďakujem konzultantovi mojej diplomovej práce, Mgr. Vladimírovi Božovi za jeho rady, trpezlivosť a pripomienky pri písaní tejto práce.

Abstrakt

Autor:	Jakub Jursa
Názov diplomovej práce:	Dátové štruktúry pre uchovávanie sekvenovacích dát
Škola:	Univerzita Komenského v Bratislave
Fakulta:	Fakulta matematiky, fyziky a informatiky
Katedra:	Katedra informatiky
Vedúci diplomovej práce:	Mgr. Tomáš Vinař, PhD.
Konzultant diplomovej práce:	Mgr. Vladimír Boža
Rozsah práce:	54 strán

Bratislava, máj 2015

V práci sa zaoberáme návrhom a implementáciou dátovej štruktúry pre efektívne indexovanie krátkych reťazcov pochádzajúcich zo sekvenovacích dát (*sequencing reads*) s možnosťou efektívneho vyhľadávania tých readov, ktoré obsahujú daný podreťazec pevne danej dĺžky. Podstatou tejto dátovej štruktúry je komprimácia vstupnej sady readov za použitia algoritmov na zostavovanie genómu (pomocou vhodného softvérového nástroja) a následná efektívna indexácia cez FM-index. Výsledná implementácia je pamäťovo výrazne efektívnejšia než existujúce riešenia pri zachovaní podobnej rýchlosti odpovedania na dotazy.

Kľúčové slová: dátové štruktúry a algoritmy, sekvenovacie ready, indexovanie, bioinformatika

Abstract

Author:	Jakub Jursa
Thesis title:	Data Structures for Storing DNA Sequencing Data
University:	Univerzita Komenského v Bratislave
Faculty:	Fakulta matematiky, fyziky a informatiky
Department:	Katedra informatiky
Advisor:	Mgr. Tomáš Vinař, PhD.
Consultant:	Mgr. Vladimír Boža
Page count:	54 pages

Bratislava, May 2015

In this work we focus on design and implementation of data structure for efficient indexing of sequencing reads with ability of searching sequence reads for pattern of given length as a substring. The efficiency of this data structure comes from compression of input sequencing reads using algorithms for genome assembly (using appropriate software tool) and consecutive efficient indexing using FM-index. Resulting implementation is much more memory efficient than existing solutions while preserving similar query time.

Keywords: data structures and algorithms, sequencing reads, indexing, computational biology

Obsah

Úvod **1**

1 Základné pojmy a algoritmy **2**

1.1 Formálne definície a označenia 2

1.2 Sekvenovanie 3

1.3 Algoritmy na zostavovanie genómu 3

1.3.1 Najkratšie spoločné nadslovo 4

1.3.2 Overlap-Layout-Consensus 4

1.3.3 De Bruijnove grafy 5

1.4 Funkcie rank a select 6

1.5 Sufixové polia 6

1.5.1 Vyhľadávanie vzorky v sufixovom poli 7

1.5.2 Konštrukcia sufixového poľa 7

1.5.3 Pamäťová efektivita 8

1.6 Burrows-Wheelerova transformácia 8

1.6.1 BWT cez BWM 9

1.6.2 BWT cez sufixové pole 9

1.6.3 Reverzná BWT s použitím LF mappingu 10

1.7 FM-index 12

1.7.1 Vyhľadávanie pomocou FM-indexu 14

2 Problém indexovania readov **17**

2.1 Definícia problému 17

2.2 Riešenie s použitím hash mapy 18

2.3 Riešenie s použitím GkArray 20

2.4 Porovnanie 21

3 CR-index **22**

3.1 Princíp fungovania 22

3.1.1 Komprimácia vstupu . . . 22
3.1.2 Index . . . 23
3.1.3 Dotazy . . . 25
3.2 Ready s chybami . . . 28
3.3 Navrhovaná implementácia . . . 30
3.3.1 Korekcia readov . . . 30
3.3.2 Budovanie indexu . . . 32
3.3.3 Dotazy . . . 34
3.3.4 Možné zlepšenia . . . 38

4 Implementácia **40**
4.1 Použité knižnice . . . 40
4.2 Inštalácia . . . 41
4.3 Vonkajšia štruktúra . . . 42
4.4 Vnútorná štruktúra . . . 44

5 Výsledky **47**
5.1 Testovacie prostredie . . . 47
5.2 Testy . . . 47
5.2.1 CR-index vs GkArray - pamäť . . . 47
5.2.2 CR-index vs GkArray - čas dotazu . . . 48
5.2.3 Pamäť vs. pokrytie . . . 50
5.3 Zhrnutie a interpretácia výsledkov . . . 51

Záver **52**

Zoznam použitej literatúry **53**

Úvod

Výsledkom príchodu nových technológií sekvenovania genómov je tlak na efektívne spracovávanie veľkého množstva dát – *sequence readov* – ktoré sekvenátory produkujú. Jedným so súvisiacich problémov je aj problém, ako efektívne tieto sequence ready indexovať a vyhľadávať v nich.

V súčastnosti nie sú k dispozícii knižnice, ktoré by tento problém vhodne riešili – buď sú príliš všeobecné alebo sú neefektívne.

V tejto práci sa zaoberáme návrhom a implementáciou dátovej štruktúry, pomocou ktorej by bolo možné efektívne indexovať ready a vyhľadávať v nich podľa vopred určených kritérií.

V prvej kapitole sa oboznámime so základnými pojmami a algoritmami. V druhej kapitole zadefinujeme *problém indexovania readov* ako algoritmickú úlohu. V kapitole č. 3 predstavíme náš návrh riešenia tohto problému – dátovú štruktúru *CR-index*. V nasledujúcej kapitole popíšeme jej implementáciu, ktorú sme realizovali v jazyku `C++` a v poslednej, piatej kapitole, prezentujeme výsledky meraní efektívnosti CR-indexu.

Kapitola 1

Základné pojmy a algoritmy

V tejto kapitole sa oboznámime so základnými pojmami z bioinformatiky a s niektorými dátovými štruktúrami a algoritmami, ktoré budú neskôr použité pri implementácii.

1.1 Formálne definície a označenia

Definícia 1.1. *Abeceda je konečná neprázdna množina symbolov (písmen). Označujeme ju* Σ.

Poznámka 1.1. V našom kontexte bude pracovať s abecedou $\Sigma = \{A, C, T, G\}$.

Definícia 1.2. *Reťazec nad abecedou* Σ *je konečná postupnosť symbolov z* Σ.

Definícia 1.3. *Dĺžka reťazca je dĺžka postupnosti, ktorá ho vytvára. Dĺžku reťazca* s *označujeme* $|s|$.

Definícia 1.4. *Podreťazcom reťazca* $s = a_0a_1 \dots a_n$ *je súvislá podpostupnosť* $t = a_ia_{i+1} \dots a_j$, *pričom platí* $0 \leq i \leq j \leq n$. *Takýto podreťazec označujeme* $t = s[i, j]$. *Špeciálne, ak* $i = 0$, *tak takýto podreťazec nazývame prefix a ak* $j = n$, *tak sufix reťazca* s.

Označenie 1.1. *I-ty symbol postupnosti tvoriacej reťazec* s *budeme označovať* $s[i]$.

Definícia 1.5. *K-merom nazývame ľubovoľný podreťazec dĺžky* k *ľubovoľného reťazca* s *z množiny reťazcov* S.

Definícia 1.6. *Reverzným komplementom reťazca* $S = a_0a_1 \dots a_{n-1}$, *kde* $\forall i \in \mathbb{N} : 0 \leq i < n : a_i \in \{A, C, T, G\}$ *je reťazec* $S_{rc} = b_0b_1 \dots b_{n-1}$, *pre ktorý platí:*

$$\forall i \in \mathbb{N} : 0 \leq i < n : b_i = \begin{cases} A : a_{n-i-1} = T \\ T : a_{n-i-1} = A \\ C : a_{n-i-1} = G \\ G : a_{n-i-1} = C \end{cases}$$

Príklad 1.1. Pre reťazec $S = ACTTTGCCCT$ je reverzným komplementom reťazec $S_{rc} = AGGGCAAAGT$.

1.2 Sekvenovanie

Sekvenovanie DNA je súhrnný termín pre biochemické metódy, pomocou ktorých sa zisťuje poradie nukleotidov (A, C, T, G) v sekvenciách DNA. Tieto metódy nedokážu prečítať celý genóm naraz, ale len po malých častiach.

Z informatického pohľadu je výsledkom sekvenovania DNA sada reťazcov nad abecedou $\Sigma = \{A, C, T, G\}$[1], pričom tieto reťazce pochádzajú z jedného spoločného nadslova. Tieto reťazce nazývame *sequencing reads*. Pri sekvenovaní nás zaujímajú hlavne:

- Dĺžka readov - tá je daná použitou biochemickou metódou sekvenovania.
- Počet readov
- Miera pokrytia - tá nam hovorí, v aspoň koľkých readoch sa každá báza sekvenovanej DNA nachádza.
- Miera chýb - žiadna sekvenovacia metóda nie je dokonalá a teda daný read nemusí byť v skutočnosti podslovom sekvenovanej DNA, ale môže obsahovať chyby, napríklad bázy navyše, či niektoré zmenené.

Ready je následne potrebné bioinformatickými metódami poskladať do dlhších fragmentov na základe ich prekryvov – tie sa nazývajú *kontigy*. Túto úlohu riešia softvérové nástroje nazývané *assemblery*.

1.3 Algoritmy na zostavovanie genómu

Výsledkom sekvenovania genómu (pri použití súčasných technológií) je veľké množstvo malých fragmentov – *sequencing reads*, ktoré je potrebné zostaviť do jednej súvislej sekvencie. Náročnosť tejto úlohy závisí najmä od:

- dĺžky zostavovaného genómu - čím je kratší, tým je to jednoduchšie
- dĺžok jednotlivých fragmentov - čím sú dlhšie, tým lepšie
- priemernej hĺbky pokrytia - tzn koľko fragmentov v priemere pokrýva konkrétnu pozíciu zostavovanej sekvencie - čím viac, tým lepšie

[1] Veci začneme zjednodušovať už hneď zhurta zo začiatku a nebudeme rátať s neznámymi - N (*uNknown*) bázami v sequencing readoch.

1.3.1 Najkratšie spoločné nadslovo

Asi najjednoduchšiu formuláciu problému zostavovania genómu predstavuje problém hľadania najkratšieho spoločného nadslova (*shortest common superstring*).

Definícia 1.7. *Najkratším spoločným nadslovom reťazcov $S_1, S_2, \ldots, S_k$ nazývame najkratší reťazec S taký, že každé S_i je podslovo S.*

Tento problém je ale NP-ťažký[GJ02], takže nepoznáme žiadny rýchly algoritmus, ktorý vždy nájde riešenie. Jednoduchá heuristika by vyzerala nasledovne:

- nájdi také dva fragmenty, ktoré majú najväčší prekryv, t.j. najdlhšiu α takú, že $S_i = \alpha\beta$ a $S_j = \gamma\alpha$
- spoj S_i a S_j do $\gamma\alpha\beta$
- opakuj prvý krok, kým nezostane iba jedno slovo

O tomto algoritme sa dá dokázať, že je to 2.5-aproximačný algoritmus[ZS00], t.j. nájdené riešenie bude mať vždy dĺžku nanajvýš 2.5-násobku optimálnej dĺžky. Síce je problém najkratšieho spoločného nadslova pomerne elegantnou formuláciou problému zostavovania genómov, no riešenia, ktoré produkuje nemusia byť stále správne – problém nastáva pri genómoch s opakujúcimi sa časťami.

1.3.2 Overlap-Layout-Consensus

Táto metóda, ako už jej názov naznačuje, pracuje v troch fázach:

Overlap

V tejto fáze assembler hľadá navzájom sa prekrývajúce fragmenty a spája ich do dlhších fragmentov, *kontigov*.

Porovnávanie každej dvojice fragmentov by však bolo v praxi veľmi časovo náročné, nakoľko fragmentov môžu byť často milióny a pri porovnaní každej dvojice by sme potrebovali rádovo $O(n^2k)$ času (kde n je počet fragmentov a $O(k)$ je čas, ktorý trvá porovnanie dvoch fragmentov).

Efektívnejšie algoritmy sú založené na predpoklade, že ak sa dva fragmenty úplne zhodujú na úseku dĺžky aspoň k (v praxi sa používa $k \approx 24$), tak pochádzajú z toho istého miesta v sekvenovanom genóme a teda sú spojené do jedného kontigu.

Vo všeobecnosti vyzerá tento postup nasledovne:

- vytvor zoznam podreťazcov dĺžky k zo všetkých fragmentov, pričom ku každému podreťazcu si zapamätaj, z ktorého fragmentu pochádza
- lexikograficky utrieď zoznam podreťazcov
- skupiny výrazne sa prekrývajúcich fragmentov sa v utriedenom zozname budú vyskytovať pohromade a práve to sú dobrí kandidáti na spojenie do kontigov

Layout

V druhej fáze assembler určuje relatívnu polohu jednotlivých kontigov a ich približné vzdialenosti, výsledné bloky kontigov sa nazývajú *superkontigy*.

Consensus

V poslednej fáze je vytvorená finálna sekvencia superkontigov, ktorá pokrýva genóm (alebo aspoň jeho časti).

1.3.3 De Bruijnove grafy

Tento prístup využíva to, že (obvykle) máme pri sekvenovaní veľké množstvo dát, a preto si môžme dovoliť časť obsiahnutej informácie v nich odignorovať.

Pre jednoduchosť budeme predpokladať, že všetky fragmenty pochádzajú z jedného vlákna a snažíme sa zostaviť len jeden, úplne pokrytý chromozóm.

Algoritmus pre vytvorenie *de Bruijnovho grafu* vyzerá nasledovne:

- vytvor zoznam všetkých podreťazcov dĺžky k zo všetkých fragmentov (k-tice)
- vrcholy grafu budú tvoriť všetky úseky dĺžky $k-1$ (teda podreťazce podreťazcov dĺžky k)
- jednotlivé k-tice budú reprezentované orientovanými hranami v grafe, pričom k-tica $s_1 s_2 \dots s_k$ spája vrcholy $s_1 s_2 \dots s_{k-1}$ a $s_2 s_3 \dots s_k$.

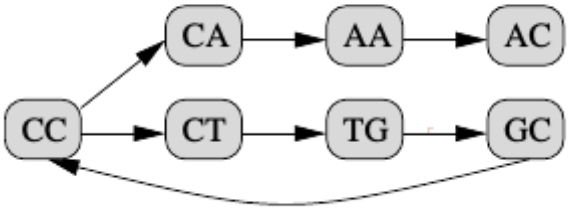
Eulerovský ťah v takomto grafe potom predstavuje hľadaný genóm.

Príklad 1.2. Uvažujme fragmenty `CCTGCC` a `GCCACC` a $k = 3$. Zoznam k-tic teda bude vyzerať nasledovne:

CCTGCC	**GCCACC**
CCT	GCC
CTG	CCA
TGC	CAC
GCC	ACC

Úseky dĺžky $k-1$: CC, CT, TG, GC, CA, AA, AC

Graf:



1.4 Funkcie rank a select

Definícia 1.8. *Funkcia rank na reťazci S je definovaná ako $rank_S(i,c) = n$, kde n predstavuje počet výskytov znaku c v reťazci $S[1,i]$. Ak $i \leq 0$, potom $rank_S(i,c) = 0$.*

Definícia 1.9. *Funkcia select na reťazci S je definovaná ako $select_S(n,c) = i$, kde i predstavuje najmenší index v reťazci S, pre ktorý platí, že počet výskytov znaku c v $S[1,i]$ je n. Funkcia select je inverzná funkcia ku funkcii rank.*

Príklad 1.3. $S = banana$. Potom $rank_S(4,a) = 2$ a $select_S(1,n) = 3$.

Nad daným reťazcom sa tieto funkcie dajú implementovať s malou pamäťovou zložitosťou a časovou zložitosťou $O(log(\sigma))$[GGV03].

1.5 Sufixové polia

Sufixové pole je jednoduchá dátová štruktúra používaná napríklad pri indexácii, kompresných algoritmoch alebo v bioinformatike. Tento koncept bol predstavený v roku 1990 [MM90]. Bol navrhnutý ako pamäťovo efektívnejšia náhrada sufixových stromov.

Definícia 1.10. *Nech $S = s_1 s_2 \dots s_n$ je reťazec a nech $S[i,j]$ označuje podreťazec reťazca S od i po j, t.j. $s_i s_{i+1} \dots s_{j-1} s_j$. Sufixové pole SA reťazca S je pole kladných čísel označujúcich začiatočné pozície sufixov reťazca S v lexikografickom usporiadaní.*

K reťazcom sa zvykne na konci pridávať špeciálny znak \$, ktorý sa v lexikografickom usporiadaní nachádza pred všetkými znakmi uvažovanej abecedy.

Príklad 1.4. Nech $S = banana\$$. Usporiadané sufixy S budú vyzerať nasledovne:

sufix	**i**
\$	7
a\$	6
ana\$	4
anana\$	2
banana\$	1
na\$	5
nana\$	3

Prislúchajúce sufixové pole: $SA = (7, 6, 4, 2, 1, 5, 3)$.

1.5.1 Vyhľadávanie vzorky v sufixovom poli

Vyhľadávanie všetkých výskytov vzorky P v texte S vlastne znamená nájsť všetky sufixy S, ktoré začínajú na P, t.j. hľadáme taký úsek od i po j v sufixovom poli SA, pre ktorý platí, že $\forall k : i \leq k \leq j : S[SA[k], |S|] = P$. Kedže je sufixové pole usporiadané, môžeme použiť binárne vyhľadávanie. V tomto prípade bude zložitosť algoritmu $O(|P| \log |S|)$. (Porovnanie dvoch reťazcov dĺžky n je realizované v čase $O(n)$). Použitím LCP (least common prefix) je možné tento čas vylepšiť na $O(|P| + \log |S|)$. Idea spočíva v tom, že pri porovnávaní reťazcov nie je potrebné niektoré znaky porovnávať znovu, ak vieme, že sú súčasťou LCP – najdlhšieho spoločného prefixu – vzorky a momentálneho prehľadávaného intervalu. V roku 2004 bol predstavený algoritmus [AKO04] pracujúci dokonca v čase $O(|P|)$.

1.5.2 Konštrukcia sufixového poľa

Klasické triedenie

Merge sort potrebuje $O(n \log n)$ porovnaní, no porovnanie každej dvojice sufixov trvá $O(n)$, takže dokopy konštrukcia sufixového poľa potrvá $O(n^2 \log n)$.

Radix sort

Triedenie pomocou radix sortu predstavuje mierne zlepšenie. Radix sort vo všeobecnosti triedi d ciferné čísla v k-árnej sústave, pričom každú cifru triedi pomocou counting

sortu. Ten utriedi n čísel z množiny $0 \dots k-1$ v čase $O(n+k)$. Celkový čas radix sortu je teda $O(d(n+k))$. Ak uvažujeme abecedu ako podmnožinu množiny $0, \dots, n-1$, tak nám čas triedenia vyjde $O(n^2)$.

Pomocou sufixového stromu

Ak najprv skonštruujeme sufixový strom (to sa dá spraviť v čase $O(n)$) a potom ho postupne prechádzame do hĺbky, pričom v každom vrchole prechádzame neprejdené hrany podľa abecedy, tak potom poradie, v ktorom navštívime listy predstavuje poradie sufixov v sufixovom poli. Nepríjemnosťou pri tomto postupe je fakt, že sufixové stromy zaberajú príliš veľa pamäte.

Lineárne algoritmy

Lineárnych algoritmov existuje niekoľko, napríklad *SA-SI* [NZC09], ktorý je momentálne jeden z najrýchlejších známych algoritmov, prípadne algoritmus od Kärkkäinena a Sandersa [KS03].

1.5.3 Pamäťová efektivita

Sufixové polia sú vo všeobecnosti efektívnejšie než sufixové stromy, no v niektorých prípadoch to nestačí. Sufixové pole vyžaduje $O(n \log n)$ bitov, no pôvodný text nad abecedou σ len $O(n \log |\sigma|)$. Pre ľudský genóm ($\sigma = \{A, C, T, G\}, n = 3.4 \times 10^9$) by teda sufixové pole zaberalo približne 16 krát viac pamäte než samotný genóm. Práve z tohto dôvodu sa objavili vylepšenia ako napríklad *komprimované sufixové polia* a *FM-index*.

1.6 Burrows-Wheelerova transformácia

Burrows-Wheelerova transformácia (BWT) je transformácia textu využívaná pri kompresii (napríklad v programe bzip2), ale aj v úsporných dátových štruktúrach na hľadanie vzorky v texte. BWT transformuje vstupný reťazec T na reťazec $BWT(T)$, ktorý je permutáciou pôvodného reťazca, no s tou vlastnosťou, že sa dá väčšinou oveľa lepšie skomprimovať než pôvodný reťazec. Táto transformácia je invertovateľná (bez potreby uloženia akýchkoľvek dát navyše), t.j. z reťazca $BWT(T)$ vieme získať pôvodný reťazec T.

1.6.1 BWT cez BWM

Postup transformácie

- na koniec reťazca T pridáme špeciálny symbol \$, ktorý sa nevyskytuje v danej abecede a zadefinujeme ho ako prvý symbol novej abecedy (vzhľadom na lexikografické usporiadanie)
- vytvoríme maticu cyklických posunov reťazca $T\$$
- lexikograficky utriedime riadky matice
- posledný stĺpec matice predstavuje $BWT(T)$

Táto lexikograficky utriedená matica cyklických posunov reťazca T sa označuje ako BWM (Burrows-Wheeler matrix).

Príklad 1.5. Uvažujme $T = banana\$$. Matica cyklických posunov a utriedená matica budú vyzerať nasledovne:

b	a	n	a	n	a	\$
a	n	a	n	a	\$	b
n	a	n	a	\$	b	a
a	n	a	\$	b	a	n
n	a	\$	b	a	n	a
a	\$	b	a	n	a	n
\$	b	a	n	a	n	a

\$	b	a	n	a	n	**a**
a	\$	b	a	n	a	**n**
a	n	a	\$	b	a	**n**
a	n	a	n	a	\$	**b**
b	a	n	a	n	a	**\$**
n	a	\$	b	a	n	**a**
n	a	n	a	\$	b	**a**

Teda $BWT(banana\$) = annb\$aa$ - posledný stĺpec utriedenej matice.

1.6.2 BWT cez sufixové pole

Medzi BWM a sufixovým poľom je zjavný súvis - pri vytváraní sufixového poľa $SA(T)$ pre reťazec T triedime sufixy reťazca T a pri vytváraní $BWM(T)$ triedime cyklické posuny T. Tento vzťah je jasnejší v príklade:

Príklad 1.6.

BWM	**SA**	**sufix** T
$banana	7	$
a$banan	6	a$
ana$ban	4	ana$
anana$b	2	anana$
banana$	1	banana$
na$bana	5	na$
nana$ba	3	nana$

Iný spôsob ako definovať $BWT(T)$ je teda cez sufixové pole $SA(T)$:

$$BWT(T)[i] = \begin{cases} T[SA[i]] & : SA[i] > 1 \\ \$ & : SA[i] = 1 \end{cases}$$

1.6.3 Reverzná BWT s použitím LF mappingu

LF mapping

Spomenuli sme, že BWT je invertovateľná operácia, no na prvý pohľad to také zrejmé určite nie je. Pripomeňme si príklad 1.5 - $BWT(banana\$) = annb\$aa$. Pri letmom pohľade čitateľ ľahko nadobudne pocit, že informácia o tom, ktoré n v $BWT(T)$ prislúcha ku ktorému n v pôvodnom reťazci T je nenávratne stratená.

BWT ale disponuje dôležitou vlastnosťou s názvom *LF mapping* (*last-to-first mapping*). Uvažujme BWM z príkladu 1.5:

$$\begin{array}{ccccccc} \$ & b & a & n & a & n & a \\ a & \$ & b & a & n & a & n \\ a & n & a & \$ & b & a & n \\ a & n & a & n & a & \$ & b \\ b & a & n & a & n & a & \$ \\ n & a & \$ & b & a & n & a \\ n & a & n & a & \$ & b & a \end{array}$$

Prepíšme T tak, že každému znaku (okrem $) dáme ako dolný index počet jeho doterajších výskytov v T: $T = b_0a_0n_0a_1n_1a_2\$$. Tomuto indexu hovoríme *rank*. Prepíšme teraz BWM s použitím *rankov*[2]:

[2]Ranky nemajú vplyv na lexikografické usporiadanie

F						**L**
\$	b_0	a_0	n_0	a_1	n_1	a_2
a_2	\$	b_0	a_0	n_0	a_1	n_1
a_1	n_1	a_2	\$	b_0	a_0	n_0
a_0	n_0	a_1	n_1	a_2	\$	b_0
b_0	a_0	n_0	a_1	n_1	a_2	\$
n_1	a_2	\$	b_0	a_0	n_0	a_1
n_0	a_1	n_1	a_2	\$	b_0	a_0

LF mapping nám hovorí nasledovnú vec: *i-ty* výskyt znaku c v F má rovnaký *rank* ako *i-ty* výskyt c v L. Všimnime si to napríklad na znaku a – v F sú ranky v poradí: 2, 1, 0, rovnako ako v L. To platí kvôli tomu, že keď si zvolíme znak c, tak usporiadanie jeho rankov v F je dané tým, čo nasleduje po tomto znaku c v pôvodnom reťazci T (pre a je to teda poradie $\$, na\$, nana\$$). To je ale to isté, čím je dané poradie znakov a v L – tiež tým, čo sa nachádza za daným znakom v pôvodnom reťazci T.

Reverzná BWT

Na začiatku teda poznáme len $BWT(T)$, čo je posledný stĺpec (L) matice lexikograficky usporiadaných cyklických rotácií pôvodného reťazca T. Pomocou neho vieme vyrátať prvý stĺpec (F) matice jednoduchým utriedením posledného stĺpca. Pri BWT sme počítali rank vzhľadom na T, teraz ho budeme počítať vzhľadom na $BWT(T)$:

F	**L**	**rank**
\$	a	0
a	n	0
a	n	1
a	b	0
b	\$	0
n	a	1
n	a	2

Vieme, že platia nasledovné veci:

1. Predchodca znaku v F je znak v L v tom istom riadku.
2. Pomocou *LF mappingu* vieme ktorý znak z L prislúcha ktorému znaku v F.

Algoritmus na *spätnú rekonštrukciu* T z $BWT(T)$ teda vyzerá nasledovne:

1. Nájdi pozíciu *pos* posledného znaku c v T[3].
2. Pomocou *LF mappingu* vypočítaj pozíciu i znaku c v F.
3. Do *pos* priraď i.
4. Nájdi predchodcu p znaku c z $BWT(T)[i]$.
5. Do c priraď p.
6. Opakuj kroky 2 až 6 pokým nie je zrekonštruovaný pôvodný reťazec T.

Pre náš príklad by postup vyzeral tak, že by sme začali v prvom riadku (resp. nultom), teda $pos = 0$, $c = a$. Ďalej by sme sa posunuli v F na riadok 1 ($i = 1$), kde sa nachádza a s rankom 0. Na konci tohto riadka je n, to je teda predposledný znak pôvodného reťazca T. V tomto kroku priradíme do c znak n a opakujeme procedúru.

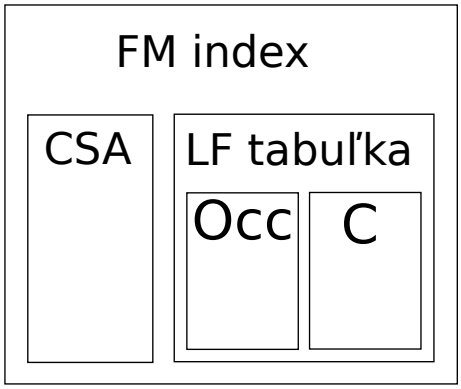
Rekonštrukcia odpredu funguje podobne, no nasledujúce znaky sa extrahujú z prvého stĺpca rovnakého riadka a na posun ďalej sa využíva *FL mapping*, ktorý je podstatne zložitejší na implementáciu.

1.7 FM-index

FM-index je komprimovaný index založený na Burrows-Wheelerovej transformácii, publikovaný v [FM00]. Skratka *FM* je odvodená od *Full-text in Minute space – full-text* index je dátová štruktúra nad textom T umožňujúca efektívne vyhľadávanie podreťazcov v T. *Minute space* znamená, že index je pamäťovo veľmi efektívny. Táto dátová štruktúra vo všeobecnosti podporuje dve operácie – operáciu *count*, ktorá pre reťazec na vstupe vráti počet výskytov v T a operáciu *locate*, ktorej výsledkom pre vstupný reťazec P je zoznam indexov v T, na ktorých sa P vyskytuje v T.

Na FM-index sa dá pozerať ako na súbor tabuliek skladajúci sa z dvoch častí s istou funkcionalitou navyše. Prvú časť tvoria tabuľky, ktoré zabezpečujú vykonanie operácie *count*. Do tejto kategórie patria všetky tabuľky podporujúce *LF mapping*. Presnejšie, táto skupina sa skladá z dvoch tabuliek - tabuľky *prefixových súm* C a *tabuľky výskytov* tiež označovanej ako Occ. Druhú kategóriu tvoria tabuľky, pomocou ktorých FM-index odpovedá na dotaz *locate*.

[3]Posledný znak T je prvý znak $BWT(T)$, kvôli pridaniu \$



Obr. 1.1: Znázornenie štruktúry FM-indexu

Komprimované sufixové pole

Komprimované sufixové pole (*CSA - compressed suffix array*) pole sa využíva na zistenie pozície daného podreťazca v texte T. Nie je si ale potrebné držať v ňom pozície všetkých sufixov T, keďže pomocou LF mappingu vieme zrekonštruovať podreťazec T začínajúci na ľubovoľnej pozícii. Preto si stačí pamätať len pozície niektorých sufixov a potom pomocou LF mappingu len čiastočne rekonštruovať T, pokým nedosiahneme nejakú pamätanú pozíciu sufixu. Komprimované sufixové pole zaberá iba zlomok pamäte oproti pôvodnému, no na oplátku sa kvôli nutnosti čiastočnej rekonštrukcie výrazne zvýši čas trvania operácie *locate*.

Tabuľka prefixových súm

Táto tabuľka obsahuje pre každý znak c danej abecedy počet výskytov lexikograficky menších znakov ako c v danom reťazci S. Narozdiel od sufixového poľa ale nie je tak ľahko komprimovateľná, keďže frekvencie výskytu jednotlivých znakov sú na sebe navzájom nezávislé. Na druhej strane, zvyčajne býva veľkosť abecedy rádovo menšia než dĺžka textu, takže veľkosť tejto tabuľky bude prispievať do celkovej pamäťovej náročnosti len malou časťou[4].

Tabuľka výskytov

Tabuľka výskytov Occ pre dané c a k vráti počet výskytov znaku c v prefixe $S[0..k]$ daného reťazca S. Keď túto tabuľku skonštruujeme nad $BWT(T)$, tak LF mapping vieme zadefinovať ako

$$LF(i) = C[BWT(T)[i]] + Occ(BWT(T)[i], i)$$

[4]Špeciálne to platí v našom, bioinformatickom kontexte, kde používame štvorpísmenovú abecedu a genómy majú dĺžky v miliónoch.

Rýchlosť prístupu k tejto tabuľke a jej pamäťová náročnosť je najdôležitejším faktorom FM-indexu.

Pre lepšiu predstavu uveďme príklad:

Príklad 1.7. Nech $T = banana\$$, potom $BWT(T) = annb\$aa$. Tabuľky C a Occ skonštruované nad $BWT(T)$ potom vyzerajú nasledovne:

C pre *annb$aa*

c	$	a	b	n
C[c]	0	1	4	5

Occ(c, k) pre *annb$aa*

	a	**n**	**n**	**b**	**$**	**a**	**a**
	0	**1**	**2**	**3**	**4**	**5**	**6**
$	0	0	0	0	1	1	1
a	1	1	1	1	1	2	3
b	0	0	0	1	1	1	1
n	0	1	2	2	2	2	2

1.7.1 Vyhľadávanie pomocou FM-indexu

Vyhľadávanie v FM-indexe rozdelíme na dve fázy – *počítaciu* a *lokalizačnú* (pri operácii *count* stačí vykonať prvú časť, pre *locate* obe). *Počítacia* fáza má za úlohu identifikovať rozsah sufixov zo sufixového poľa, ktoré majú rovnaký prefix – vyhľadávanú vzorku[5] a *lokalizačná* fáza potom identifikuje vzorky v sufixovom poli.

Počítacia fáza

Hľadanie vzorky v sufixovom poli je časovo veľmi efektívne, pretože sa zároveň hľadajú všetky výskyty danej vzorky P – simultánnym posúvaním ukazovateľov hornej a dolnej hranice v sufixovom poli. Označme tieto ukazovatele *sp* a *ep* (z ang. *starting* resp. *ending pointer*). Znak c bude obsahovať momentálne spracovávaný znak vzorky P, pričom začneme posledným znakom a budeme tiež využívať pomocné polia Occ a C zadefinované v predchádzajúcej časti. Ukazovateľ *sp* bude inicializovaný na $C[c]$ a ukazovateľ *ep* na $C[c+1]$ – v tomto prípade už tieto ukazovatele ukazujú na prvý resp. posledný výskyt posledného písmena P v F[6].

[5] Z definície sufixového poľa vyplýva, že sufixy s rovnakým prefixom nasledujú v sufixovom poli za sebou

[6] Pripomeňme, že F je prvým stĺpcom BWM a F je vlastne pole znakov, ktoré zodpovedajú indexom sufixového poľa do pôvodného textu T.

```
1  i = P.length - 1
2  c = P[i]
3  sp = C[c]
4  ep = C[c + 1]
5
6  while i > 0 do
7    i = i - 1
8    c = P[i]
9    sp = C[c] + Occ(c, sp - 1)
10   ep = C[c] + Occ(c, ep) - 1
11 end
```

Listing 1.1: Algoritmus na hľadanie vzorky pomocou FM-indexu

$Occ(c, sp-1)$ (riadok 5) vracia počet znakov c v $BWT(T)[0..(sp-1)]$. Keď k nemu pripočítame počet znakov, ktoré sú menšie než c ($C[c]$) dostávame pozíciu prvého c v rozsahu sp až ep v F. Operácie v riadkoch 5 a 6 realizujú LF mapping pre prvý a posledný výskyt c v $BWT(T)[sp..ep]$ (ak by sme nemali k dispozícii tabuľku Occ, museli by sme najprv nájsť prvý a posledný výskyt c v $BWT(T)[sp..ep]$ a až potom použiť LF mapping).

Príklad 1.8. Postup vyhľadávania vzorky $P = nan$ by vyzeral nasledovne:

$P = na\boldsymbol{n}$			$P = n\boldsymbol{an}$			$P = \boldsymbol{nan}$		
F		**L**	**F**		**L**	**F**		**L**
$	...	a	$	...	a	$	...	a
a	...	n	a	...	n	a	...	n
a	...	n	**a**	...	n	a	...	n
a	...	b	**a**	...	b	a	...	b
b	...	$	b	...	$	b	...	$
n	...	a	n	...	a	**n**	...	a
n	...	a	n	...	a	n	...	a

Lokalizačná fáza

Výsledkom počítacej fázy sú dva indexy, sp a ep, ktoré vymedzujú rozsah v sufixovom poli textu T. A práve prvky prvky v tomto rozsahu nám prezradia, na ktorých pozíciach textu T nastala zhoda s vyhľadávanou vzorkou.

Časová zložitosť

Časová zložitosť počítacej fázy je lineárna vzhľadom na dĺžku hľadanej vzorky P, keďže v každej iterácii spracujeme jeden znak P. Na druhej strane, dĺžka textu T na túto operáciu vplyv nemá, preto je táto metóda mimoriadne vhodná na spracovávanie dlhých textov, ako sú napríklad genómy. Zložitosť počítacej fázy je tiež nezávislá na počte výskytov vzorky v texte, keďže hľadáme len hornú a dolnú hranicu výskytu. V každej iterácii tiež spravíme dva dotazy do tabuľky *Occ*, takže výsledný čas závisí tiež lineárne od časovej zložitosti dotazov na *Occ*.

Pri časovej zložitosti lokalizačnej fázy je situácia opačná – nezávisí na dĺžke hľadanej vzorky, ale na počte výskytov, keďže pre každý jeden musíme pristúpiť ku sufixovému poľu. Preto o zložitosti rozhoduje implementácia sufixového poľa.

Zlepšenia

Ak by mala tabuľka *Occ* odpovedať v konštantnom čase, zaberala by značné množstvo pamäte, preto sa v praxi pri jej implementácii používajú sofistikovanejšie dátové štruktúry ako napríklad *wavelet tree* [GGV03].

Čo sa týka lokalizačnej fázy, tak je potrebné spomenúť, že pamätanie si celého sufixového poľa by zaberalo priveľa pamäte, preto sa v praxi používa napríklad komprimované sufixové pole – ukladá sa len časť pôvodného sufixového poľa a chýbajúce hodnoty sa potom v prípade potreby dorátavajú pomocou pamätaných hodnôt.

Kapitola 2

Problém indexovania readov

Indexovanie readov sa využíva v assembleri GAML[BBV14] a v nástroji CRAC[PSCR13], ktorý sa využíva na analyzovanie RNA readov.

2.1 Definícia problému

Úlohou bude teda vytvoriť efektívnu dátovú štruktúru, ktorá načíta veľkú sadu relatívne krátkych reťazcov – *sequencing reads* (pozri časť 1.2) a umožní dostatočne rýchlo odpovedať na dotaz *„vráť tie reťazce, ktoré obsahujú ako podreťazec reťazec P“*, pričom dĺžka reťazca$P$ je dopredu daná. Dôraz budeme klásť ako na rýchlosť odpovedania na dotaz, tak aj na pamäťovú efektívnosť tejto dátovej štruktúry. Čiže:

Vstup
: Na vstupe pre konštrukciu dátovej štruktúry je prirodzené číslo k - dĺžka dotazu a množina R, ktorá predstavuje množinu n readov, každý s dĺžkou l.

Výstup
: Výstupom pre dotaz p je množina S takých readov z R, ktoré obsahujú p ako podreťazec.

V našom kontexte ale platia aj nasledovné veci, ktoré nám riešenie úlohy do značnej miery uľahčia:

- Vieme, že všetky ready pochádzajú zo spoločného nadslova – sekvenovanej DNA, pričom pri sekvenovaní mohla s istou pravdepodobnosťou nastať chyba (pozri časť 1.2). Z chýb budeme uvažovať iba substitúciu, ktorej pravdepodobnosť vymedzíme na úroveň $0.1\% - 2\%$. To znamená, že pre každú bázu každého *readu* nastala substitúcia (za nie nutne rôznu bázu) s danou pravdepodobnosťou.

- Dĺžka spoločného nadslova sa pohybuje medzi miliónom (dĺžka genómu baktérií je niekde na úrovni štyroch miliónov) a jednej miliardy (dĺžka genómu človeka je asi tri miliardy báz).
- Dĺžky readov l sa pohybujú v rozmedzí $100 - 150$ báz.
- Pri sekvenovaní sa využíva miera pokrytia (pozri časť 1.2) v rozmedzí $10\times$ až $100\times$, z čoho nám v kombinácii s dĺžkou spoločného nadslova a dĺžkou readov vychádza obmedzenie pre počet readov na vstupe na $n \in [10^5, \frac{2}{3} \cdot 10^9]$.
- Dĺžku dotazu p budeme uvažovať v rozmedzí $13 - 15$.
- A na záver, pri zisťovaní, či p je podreťazcom r budeme testovať aj to, či reverzný komplement (pozri časť 1.1) p ($revcompl(p)$) nie je podreťazcom r – bude nám stačiť, ak bude túto podmienku spĺňať jeden z nich.

Cieľom bude dosiahnuť čo najnižšiu pamäťovú zložitosť, pri zachovaní „rozumnej“ časovej zložitosti. Očakávaná pamäťová zložitosť bude teda $O(n + s)$, kde n je počet načítaných readov a s je dĺžka spoločného nadslova. (Triviálnym riešením by bolo $O(n \cdot L)$, kde L je dĺžka readu)

2.2 Riešenie s použitím hash mapy

Ako prvé netriviálne riešenie tohto problému sa naskytá použitie hash mapy, kde kľúčom sú všetky možné hľadané vzorky p (tie generovať vieme, keďže máme dopredu danú dĺžku $k = |p|$ a pracujeme nad konečnou abecedou) a hodnota pre daný kľúč by bol zoznam readov, ktoré túto vzorku obsahujú. Algoritmus na generovanie takejto hash mapy by vyzeral naslednovne:

```
1  h = HashMap.new
2
3  def process(R)
4    foreach r : R do
5      for i in (0...(|r| - k + 1)) do
6        t = r[i, k]
7        if not h.has_key?(t)
8          h[t] = LinkedList.new
9        end
10
```

```
11        h[t].append(r)
12      end
13    end
14  end
15
16  def find_reads(p)
17    return (h[p].append(h[revcompl(p)]))
18  end
```

Listing 2.1: Algoritmus na riešenie problému indexovania readov pomocou hash mapy

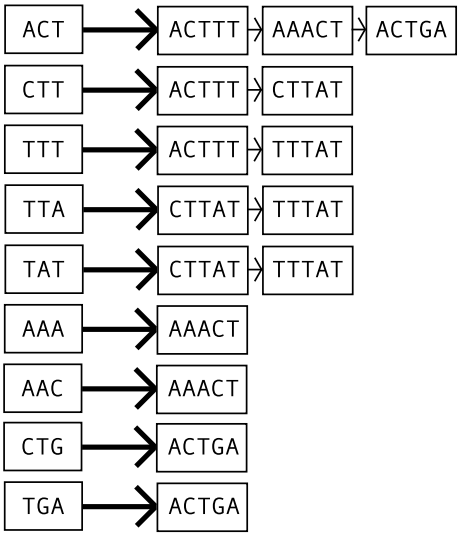
Pre každý read r zo zoznamu readov R vygenerujeme všetky jeho podreťazce dĺžky k a tie použijeme ako kľúče do hash mapy, pomocou ktorých tento read zaindexujeme jeho pridaním do spájaného zoznamu.

Hľadanie vzorky p potom prebieha tak, že vrátime zreťazené spájané zoznamy pre p a reverzný komplement p.

Príklad 2.1. Príklad hash mapy pre množinu readov

$$S = \{ACTTT, CTTAT, TTTAT, AAACT, ACTGA\}$$

.



Obr. 2.1: Hash mapa

Pre vzorku napr. TTT je výsledkom množina *readov* $R = \{ACTTT, AAACT\}$. $AAACT$ sa do výsledku dostane kvôli tomu, že hľadáme aj reverzné komplementy.

Vyhľadávanie vzorky prebieha v konštantnom čase, no pamäťová zložitosť tohto riešenia ale nie je ani zďaleka optimálna, v najhoršom prípade až $O(n \cdot l)$. Existuje množstvo zlepšení, napríklad:

- miesto celých readov nám stačí si v spájanom zozname pamätať len indexy daných readov
- kľúče možno komprimovať napríklad nasledovným spôsobom: pre znaky tvoriace kľúče vytvoríme nasledovné kódovanie: $A = 00$, $C = 01$, $T = 10$, $G = 11$ a pomocou neho zakódujeme celý kľúč, čím dostaneme bitový vektor (v našom príklade uvedenom vyššie 6-bitový), ktorý sa už komprimuje oveľa jednoduchšie (napríklad ako celé číslo).

2.3 Riešenie s použitím GkArray

V roku 2011 vyšiel v *BMC Bioinformatics* článok od N. Phillipea a spol.[NP11] prezentujúci dátovú štruktúru s názvom *GkArray*. Cieľom autorov bolo navrhnúť a implementovať univerzálne riešenie na indexáciu readov a vyhľadávanie v nich cez k-mery vopred danej dĺžky. Vstupom pre jej konštrukciu je sada readov, každý dĺžky m, a číslo k, kde $k \leq m$. Dátová štruktúra *GkArray* je konštruovaná tak, aby vedela odpovedať na nasledujúce dotazy[1]:

Q1: Ktoré ready obsahujú f ako podreťazec?

Q2: Koľko readov obsahuje f ako podreťazec?

Q3: Na ktorých pozíciach sa v jednotlivých readoch f nachádza?

Q4: Aký je celkový počet výskytov f vo všetkých readoch?

Q5: V ktorých readoch sa f nachádza len raz?

Q6: V koľkých readoch sa f nachádza len raz?

Q7: Na ktorých pozíciach v readoch, ktoré ho obsahujú len raz sa f nachádza?

GkArray vnútorne používa 4 polia – *GkSA* (Generalized k Suffix Array), čo je vhodným spôsobom modifikované sufixové pole skonštruované nad reťazcom získaným zreťazením všetkých readov zo vstupu, *GkIFA* (Generalized k Inverse Array) predstavuje

[1] V pôvodnom článku je uvedený aj ôsmy dotaz, ktorý je ale ekvivalentný so šiestym, my sme si ho dovolili rovno vynechať.

obmenu sufixového poľa reverzov, *GkCFA* (Generalized k Counting Factor Array) a štvrté pole, *GkCFPS* (Generalized k Counting Factor Prefix Sum) je použité len pri konštrukcii prvých troch.

Vidíme teda, že štruktúra *GkArray* by mala vedieť riešiť *problém indexovania readov* podľa našej definície pomocou odpovedí na dotazy **Q1** – **Q4**. Pri indexácii ale len zreťazí reťazce zo vstupu a nijak nevyužíva to, že sú to sekvencieho nejakého genómu, nesnaží sa ich nijak komprimovať. Skúsme teda spraviť jednoduché porovnanie spotreby pamäte v porovnaní s hash mapou.

2.4 Porovnanie

Pre porovnanie sme použili náhodne vybraných milión readov z *E.Coli MG1655 Illumina HiSeq2000 sequencing dataset*[2], pričom dĺžka každého readu bola 151 báz a dĺžku dotazu sme zvolil $k = 13$. Program, ktorý sme použili na testovanie je dostupný na internete na adrese `https://github.com/kuboj/CR-index/blob/master/benchmark/construct.cpp`.

Kompilácia a test prebehli počítači s nainštalovanou 64-bitovou verziou operačného systému Linux 3.16.0-33 a kompilátorom `gcc` verzie 4.9.1. Pri kompilácii boli zapnuté všetky optimalizačné prepínače (`-O3`).

Ako implementáciu hash mapy sme zvolili `stl::unordered_map <string, vector<int>>` - to znamená, že sme nepamätáme všetky ready, ale len ich indexy a tie vypisujeme na výstup . Nepoužili sme ani komprimáciu kľúčov, ktorú sme vyššie naznačili.

Vo výsledku zaberala hash mapa 2703080kB a *GkArray* 1241084kB (pričom čas konštrukcie hash mapy bol len o málo kratší). Keď si uvedomíme, že vstup tvorilo 150MB dát, tak nám pre *GkArray* vychádza „cena“ za jednu bázu na vstupe veľmi vysoká, okolo 8 bajtov. Určite si preto myslíme, že je v tejto oblasti priestor na zlepšenie.

[2]Dostupné verejne na internete na `ftp://webdata:webdata@ussd-ftp.illumina.com/Data/SequencingRuns/MG1655/MiSeq_Ecoli_MG1655_110721_R1.fastq`

Kapitola 3

CR-index

V tejto kapitole predstavíme naše riešenie *problému indexovania readov* – dátovú štruktúru *CR-index* (z eng. *Compressed Read index*), ktorá je navrhnutá tak, aby sa vyhla nedostatkom, ktorými trpia riešenia s použitím *GkArray* alebo hash mapy. Pripomeňme, že úlohou tejto dátovej štruktúry bude efektívne (najmä vzhľadom na pamäť) zaindexovať veľkú sadu relatívne krátkych reťazcov nad malou abecedou – *sequence readov* – a má vedieť odpovedať na dotaz *„vráť mi všetky ready také, že obsahujú p ako podreťazec“*.

3.1 Princíp fungovania

3.1.1 Komprimácia vstupu

Kým *GkArray* ready zo vstupu iba zreťazil a nad týmto dlhým reťazcom potom budoval sufixové polia a ostatné pomocné štruktúry, hlavnou myšlienkou *CR-indexu* je, ako už napovedá názov, ready najprv vhodným spôsobom skomprimovať a až potom ďalej spracovávať. Využijeme to, že reťazce, ktoré máme za úlohu indexovať majú čosi spoločné – vznikli sekvenovaním jedného genómu. Keďže pokrytie sa pri sekvenovaní zvyčajne pohybuje na úrovni $10\times$ – $100\times$, znamená to, že ready obsahujú veľké množstvo nadbytočných informácií. Pomocou vhodného assembleru čiastočne zrekonštruujeme pôvodný genóm – čiastočne preto, lebo výstupom assemblerov pre nejakú sadu readov nebýva vo všeobecnosti súvislý genóm, ale len sada kontigov resp. superkontig. Tiež nemáme zaručené, že každý read zo vstupu sa v niektorom kontigu musí nachádzať. Tieto chýbajúce ready potom musíme efektívne nájsť a nejakým spôsobom pridať ku kontigom, aby sme neprišli o žiadnu informáciu.

Pre začiatok predpokladajme, že ready na vstupe sú bez sekvenovacích chýb – to

znamená, že pre každý read platí, že je podreťazcom pôvodného genómu, ktorý bol sekvenovaný. Assembler by mal byť pri rekonštruovaní genómu pomerne úspešný (samozrejme ak majú ready dostatočné pokrytie) a relatívne málo readov by malo chýbať medzi kontigmi. V tomto prípade by sme teda spolu zreťazili kontigy a všetky chýbajúce ready – tým dostaneme bezstratovo komprimovaný vstup. Ako ale nájdeme ready, ktoré nie sú podreťazcom zreťazenia kontigov? Jednoducho – nad zreťazením kontigov skonštruujeme FM-index, ktorého sa postupne budeme pýtať na všetky ready zo vstupu (a ich reverzné komplementy) a v prípade, že FM-index tento read (ani jeho reverzný komplement) nenájde, označíme ho ako chýbajúci.

Výsledkom komprimačnej fázy je teda zreťazenie kontigov a readov zo vstupu, ktoré nie sú podreťazcom zreťazenia kontigov a ani ich reverzný komplement nie je podreťazcom zreťazenia kontigov. Tento dlhý reťazec nazveme *superstring* (neskôr ho definujeme aj formálne).

Superstring v princípe predstavuje najkratšie spoločné nadslovo pre ready, no keďže hľadanie najkratšieho spoločného nadslova je NP-úplný problém, musíme sa uchýliť ku vhodnej aproximácii, čo za nás spraví assembler.

3.1.2 Index

V ďalšej fáze konštrukcie CR-indexu vytvoríme nad superstringom (opäť) FM-index. V tejto chvíli sme už schopní efektívne vyhľadávať vzorky v superstringu, no keď nám operácia FM-indexu *locate* vráti zoznam pozícií superstringu, kde sa daná vzorka nachádza, nevieme z toho povedať ktorým readom tieto pozície prislúchajú. Potrebujeme si teda predrátať, kde sa ktorý read nachádza. Na to použijeme pole *positions* – jeho prvkami budú trojice (i, r, b) - i bude označovať index v superstringu, kde začína read r. Logická premenná b označuje, či sa v superstringu nachádza samotný read ($b = 0$) alebo jeho reverzný komplement ($b = 1$). Algoritmus na konštrukciu CR-indexu teda vyzerá nasledovne[1].

```
1 assembler = Assembler.new
2 superstring = ""
3 positions = Array.new
4
5 contigs = assembler.assemble(R)
6 foreach c : contigs do
```

[1] Táto implementácia slúži len pre účely vysvetlenia konštrukcie CR-indexu. Reálna implementácia sa do istej miery odlišuje a popíšeme ju neskôr.

```
7   superstring += c
8 end
9
10 contigs_fm_index = FMIndex.new(joint_contigs)
11
12 foreach r : R  do
13   matches = contigs_fm_index.locate(r)
14   matches2 = contigs_fm_index.locate(rev_compl(r))
15
16   if matches.size() == 0 && matches2.size()
17     positions.push(superstring.length(), r, 0)
18     superstring += r
19   else
20     foreach m : matches do
21       positions.push([m, r, 0])
22     end
23
24     foreach m : matches2 do
25       positions.push([m, r, 1])
26     end
27   end
28 end
29
30 positions.sort
31 fm_index = FMIndex.new(superstring)
```

Listing 3.1: Algoritmus konštrukcie CR-indexu nad readmi bez chýb.

Premenná R označuje množinu readov, ktorú máme na vstupe. Objekt *assembler* (riadok 1) predstavuje zapuzdrenie volania vhodného assemblera, jeho metóda *assemble* (riadok 5) vracia pre pole readov na vstupe pole reťazcov predstavujúcich kontigy, ktoré assembler poskladal z readov. Objekt *FM-index* (riadok 10) je vhodná implementácia FM-indexu podporujúca operáciu *locate* (viď. kapitolu 1.7), ktorá pre vzorku na vstupe vráti pole indexov reťazca nad ktorým je tento FM-index skonštruovaný, na ktorých daná vzorka začína. Funkcia *rev_ compl(s)* (riadok 14) vracia pre reťazec na vstupe jeho reverzný komplement (podľa definície 1.6).

Algoritmus najprv zavolá assembler (riadok 5), pomocou ktorého vytvorí kontigy, ktoré zreťazí (riadky 6 – 8). Potom prebehne konštrukcia pomocného FM-indexu (ria-

dok 10) nad zreťazenými kontigmi. Následne sa pre každý read r a jeho reverzný komplement zavoláme metódu *locate* tohto FM-indexu (riadky 13 – 14). Ak sa ani jeden z nich v superstringu nenachádza, tak ho tam pridáme (riadok 18). Už predtým už ale vieme, na akej pozícii bude tento read v superstringu začínať (keďže ho pridávame na koniec) a môžeme túto informáciu vložiť do poľa *positions* (riadok 17). Pre ready, ktoré sa v superstringu nachádzajú pridáme do poľa *positions* informáciu o ich výskytoch (riadok 21) resp. výskytoch ich reverzných komplementov (riadok 25). Pole *positions* nakoniec utriedime (riadok 30) – ako uvidíme v ďalšej časti, pomôže nám to rýchlejšie identifikovať ready obsahujúce hľadanú vzorku. Nad superstringom, ktorý sme úspešne zreťazili so všetkými chýbajúcimi readmi potom skonštruujeme nový FM-index (riadok 31). Práve ten spolu s poľom *positions* predstavujú výstup konštrukcie celého CR-indexu.

3.1.3 Dotazy

Úlohou je pre reťazec p na vstupe vrátiť tie ready, ktoré obsahujú p ako podreťazec. Najprv uvedieme algoritmus v pseudokóde a ten následne vysvetlíme:

```
1  def locate(p)
2    retval = Array.new
3    indexes = fm_index.locate(p)
4
5    foreach i : indexes do
6      start_index = [i + k - l, -1, 0]
7      end_index = [i, INT_MAX, 1]
8
9      low = positions.lower_bound(start_index)
10     up = positions.upper_bound(end_index)
11
12     for (pos = low; pos != up; pos++) do
13       if (pos[2] == 0)
14         retval.push(pos[1])
15       else
16         retval.push(rev_compl(pos[1]))
17       end
18     end
19   end
20
```

```
21    return retval
22  end
```

Listing 3.2: Algoritmus dotazu *locate* CR-indexu nad readmi bez chýb.

Premenné fm_index a *positions* sú výsledkom konštrukcie z časti 3.1.2, premenná l predstavuje dĺžku každého readu, premenná k má hodnotu dĺžky dotazu (podľa toho, ako sme problém *indexovania readov* zadefinovali v časti 2.1 máme obe hodnoty k dispozícii na vstupe pri konštrukcii CR-indexu, t.j. predstavujú parametre konštrukcie) a konštanta INT_MAX je najväčšia hodnota pre celočíselný dátový typ.

V prvom kroku vyhľadáme výskyty p v superstringu pomocou FM-indexu (riadok 3), ktorý sme skonštruovali počas predchádzajúcej fázy – výsledkom bude zoznam pozícií v superstringu, kde bola vzorka p nájdená. Následne potrebujeme tieto pozície preložiť na ready, ktorým prislúchajú a na to využijeme pole *positions*. To spravíme tak, že v ňom pre každý index i (riadok 5) pomocou binárneho vyhľadávania obmedzíme rozsah, kde sa nachádzajú relevantné hodnoty – tie prvky, ktoré označujú začiatky readov, ktoré obsahujú ako podreťazec tento výskyt hľadanej vzorky. Práve tu využijeme to, že toto pole je utriedené podľa prvej súradnice – indexu ukazujúceho na pozíciu v superstringu. Metódy *lower_bound*[2] (riadok 9) resp. *upper_bound* (riadok 10) vrátia takú pozíciu v poli, že hodnota na tejto pozícii nie je menšia resp. je väčšia ako $start\_index$ resp. end_index. Pre $start\_index$ preto zvolíme hodnotu, ktorá popisuje najľavejší index v superstringu taký, že read na ňom začínajúci môže obsahovať výskyt vzorky p začínajúci na indexe i (pre end_index analogicky najpravejší read. Viď príklad 3.1). Všetky takto ohraničené hodnoty poľa *positions* (riadok 12) predstavujú ready, ktoré obsahujú tento výskyt vzorky p – posledné, čo ostáva je pridať do výstupného poľa *retval* tento read (riadok 14) resp. jeho reverzný komplement (riadok 16) – podľa toho, ktorý z nich je v superstringu uložený (a indikátor toho sme si pri konštrukcii poľa *positions* uložili ako tretiu hodnotu pre každý prvok). Metóda *locate* teda pre hľadanú vzorku p vráti zoznam readov, ktoré ju obsahujú ako podreťazec. Pre splnenie poslednej podmienky z časti 2.1 by sme pre dotaz na vzorku p mali zavolať metódu *locate* aj pre vstup $rev\_compl(p)$ a výstupy pre obe volania spojiť do jedného poľa a vrátiť ako výstup to.

Príklad 3.1. Ilustrácia toho, ako by mohol vyzerať najľavejší a najpravejší read v superstringu pre vzorku $p = GTCG$, $k = 3$, $l = 10$. Ak vzorka p začína v superstringu na indexe i, tak najľavejší read začína na indexe $i + k - l$ a najpravejší na indexe i.

[2]Podľa `http://www.cplusplus.com/reference/algorithm/lower_bound/`

$$ss = \ldots ACTTG\overbrace{ATACCG\,\mathbf{GTCG}}^{read1}\underbrace{AAAAAA}_{read2}GGTC\ldots$$

Príklad 3.2. Uvažujme sadu readov $R = \{AAATTG, AATTGG, ATTGGC, GCCCAA, CCCATA, GGTAAT, GTAATC, TAATCA, ATCAAA\}$, kde $l = 6$ a $p = 4$. Ďalej predpokladajme, že výstupom assemblera sú kontigy $c_0 = AAATTGGCCCAA$ a $c_1 = TTGATTACC$. Ich zreťazením a doplnením chýbajúcich reťazcov dostávame nasledovný superstring:

$$ss = \overbrace{AAATTGGCCCAA}^{c_1}\underbrace{TTGATTACC}_{c_2}\overbrace{CCCATA}^{r_3}\underbrace{ATCAAA}_{r_7}$$

Pole *positions* potom vyzerá nasledovne:

i	read	rev_compl?
0	AAATTG	0
1	AATTGG	0
2	ATTGGC	0
6	GCCCAA	0
13	TAATCA	1
14	GTAATC	1
15	GGTAAT	1
21	CCCATA	0
27	ATCAAA	0

Ilustrujme dotaz pre vzorku $p = AAT$:

$$A\mathbf{AAT}TGGCCCAATTGATTACCCCCCAT\mathbf{AAT}CAAA$$

Vidíme, že FM-index nad superstringom nám ako výsledok vrátil dve hodnoty – 1 a 26. Ak $i = 1$, potom začiatok najľavejšieho readu, ktorý môže vzorku obsahovať je 0, začiatok najpravejšieho 1, iterátor *low* (riadok 9 listingu 3.2) ukazuje na index 0 poľa *positions* a iterátor *up* (riadok 10) na pozíciu 2 (túto hodnotu už iterátor *pos* pri iterovaní (riadok 12) nenabudne). Výsledkom teda budú ready začínajúce na indexoch 0 a 1 – *AAATTG* a *AATTGG*.

Pokračujme pre $i = 26$. Iterátor *low* bude ukazovať na index 8 v poli *positions* (lebo najľavejší read obsahujúci tento výskyt vzorky p môže začínať na indexe 23). Iterátor

up bude ukazovať tiež na index 8, takže cyklus *for* na riadku 12 sa nevykoná ani raz – čo je v poriadku, keďže síce sme našli výskyt vzorky v superstringu, no nachádzal sa akurát na rozhraní readov a teda ani jeden read ho neobsahoval.

V tomto momente je dôležité si všimnúť, že týmto spôsobom sme nenašli všetky výskyty vzorky $p = AAT$, napríklad read $GGTAAT$ ju obsahuje, no tento read sme (zatiaľ) na výstup neposunuli. Assemblery totiž pokladajú ready a ich reverzné komplementy za identické a teda sa môže stať, že niektorý kontig bude zložený len z reverzných komplementov – v našom príklade je to c_1. Preto pri zisťovaní výskytu vzorky p potrebujeme vykonať dopyt aj pre $rev\_compl(p)$ a výsledné množinu readov zjednotiť. Pre vzorku $TAA = rev\_compl(AAT)$ nám už FM-index vráti výskyty v superstringu, ktoré potom úspešne preložíme na chýbajúce ready.

Je nutné poznamenať, že realizáciou dotazovania týmto spôsobom vrátime aj ready, ktoré obsahujú reverzný komplement vzorky p – predstavme si, že by sme robili dotaz pre $p = TACC$, v tom prípade vrátime na výstup aj read $GGTAAT$, ktorý túto vzorku neobsahuje. To je ale v poriadku, keďže to súhlasí s našou definíciou *problému indexovania readov* (viď poslednú odrážku v časti 2.1).

3.2 Ready s chybami

Bohužiaľ, zjednodušením situácie vo forme popierania existencie sekvenovacích chýb sa až príliš vzďaľujeme realite. Ako sa teda zmení situácia, ak máme brať do úvahy, že v readoch sa vyskytujú chyby?

Pripomeňme, že pri probléme *indexovania readov* tak, ako sme ho zadefinovali v kapitole 2 uvažujeme len substitučné chyby – čo znamená, že read sa môže v niektorých bázach líšiť od svojho „obrazu“ v pôvodnom genóme.

V prvom rade assembler nebude pri konštrukcii kontigov ani zďaleka tak úspešný ako v predchádzajúcom prípade – chyby budú znižovať pravdepodobnosť, že úspešne odhalí prekryvy medzi readmi, čiže výstupom assembleru budú oveľa kratšie kontigy, čiže aj „kompresný pomer“ bude výrazne nižší a to viac readov bude potrebné pripojiť na koniec. Ak si uvedomíme, že pri indexovaní readov napríklad s dĺžkou 100 báz a chybovosťou 1% má takmer každý read aspoň jednu chybu, vidíme, že tento spôsob indexovania je pre ready s chybami nepostačujúci a komprimáciu vstupu je potrebné robiť sofistikovanejšie.

„Komprimovateľnosť“ readov zvýšime tak, že pred tým, ako ich dostane assembler na vstup budú opravené – korekcia readov (*sequencing read correction*) je známy problém, na ktorý existuje množstvo riešení[KSS10] [IFI11]. Čo nám ale skomplikuje situáciu je

fakt, že kontigy budú obsahovať opravené ready a nie originálne, ktoré sme dostali na vstupe – preto si budeme musieť pamätať aj ďalšie informácie týkajúce sa aplikovaných korekcií, aby sme vedeli prevádzať transformácie medzi opravenými a neopravenými readmi. Tým sme však stále nevyriešili situáciu, čo s chýbajúcimi readmi. Ak by sme v zreťazených kontigoch hľadali pôvodné, neopravené ready, chýbalo by ich, pochopiteľne, veľmi veľa a tým pádom by komprimácia nebola tak úspešná. Zvoľme opačný prístup – v zreťazených kontigoch budeme hľadať opravené ready a tie, ktoré v kontigoch nie sú obsiahnuté zreťazíme na koniec. Zamyslime sa teraz, ako by prebiehal dotaz – ak existuje výskyt vzorky p v superstringu, znamená to, že nejaký *opravený* read obsahuje hľadanú vzorku. To ale neznamená, že ju obsahoval aj pôvodný read – našťastie túto situáciu vieme ľahko vyriešiť pomocou zapamätaných korekcií – k danému opravenému readu skonštruujeme pôvodný a v ňom overíme, či je p jeho podreťazcom. Týmto spôsobom vieme vylúčiť falošné detekcie vzorky (*false positives*). Problematickejšie to bude z druhej strany – vieme ale takto nájsť *všetky* výskyty vzorky p medzi pôvodnými readmi? Uvažujme nasledovný príklad:

Príklad 3.3. Uvažujme sadu readov[3] $R = \{AAAAACCC, TAAAACCC, AAAACCCC, AAGACCCG, CCCCTGTG\}$, kde $l = 8$ a $p = 6$. Nech korekcie readov vyzerajú nasledovne:

$$AAAAACCC \rightarrow AAAAACCC$$
$$\mathbf{T}AAAACCC \rightarrow \mathbf{A}AAAACCC$$
$$AAAACCCC \rightarrow AAAACCCC$$
$$AA\mathbf{G}ACCC\mathbf{G} \rightarrow AA\mathbf{A}ACCC\mathbf{C}$$
$$CCC\mathbf{T}TGTG \rightarrow CCC\mathbf{C}TGTG$$

Ďalej predpokladajme, že výstupom assemblera spusteného nad opravenými readmi je jeden kontig $c_0 = AAAAACCCC$. Po doplnení chýbajúcich *opravených* readov je výstupom superstring $ss = AAAAACCCCCCCCTGTG$. Zamyslime sa teraz, ako by vyzeral dotaz na vzorku $p = TAAAAC$ – tá sa nachádza v pôvodnom reade $TAAAACCC$, no v superstringu sa už nevyskytuje. Možnosťou by bolo skúsiť vyhľadať všetky vzorky líšiace sa oproti p v jednom znaku, akoby simulovať opravu chyby readu aj vo vzorke. Tým nám lineárne narastie počet dotazov na FM-index nad superstringom. Ak by sme hľadali vzorku $p = GACCCG$ (nachádza sa v pôvodnom reade $AAGACCCG$), potrebovali by sme hľadať ďalej – výskyt tejto vzorky v pôvodnom reade postihli až

[3] Táto sada readov je zámerne extrémne jednoduchá, našim cieľom je len ilustrovať princíp problému

dve korekcie. Ak by sme mali vygenerovať všetky možné vzorky líšiace sa od pôvodnej na dvoch miestach, je ich už kvadraticky veľa. Vyzerá to teda, že ani tento prístup, použiť len opravené ready, nebude sláviť úspech. Problematické vyzerajú byť tie ready, v ktorých sú pozície opráv príliš blízko seba – tak, že by mohli „zasiahnúť“ vzorku, t.j. sú bližšie ako p.

V konečnom dôsledku teda vidno, že vhodný prístup leží kdesi uprostred možností zreťaziť s kontigmi (nad opravenými readmi) pôvodné a opravené ready – budeme sa musieť vhodne vysporiadať s readmi, ktoré majú korekcie „príliš blízko seba“.

3.3 Navrhovaná implementácia

Kým predchádzajúca časť bola skôr určená na ilustráciu fungovania princípov stojacich za CR-indexom, v tejto časti už konkrétne popíšeme jeho vnútorný dizajn.

3.3.1 Korekcia readov

Korekčná funkcia

Označenie 3.1. *Označme ako $corr_R(r)$* korekčnú *funkciu nad množinou readov R, ktorej výstupom pre read r je opravený read r_{corr}, čo je read rovnakej dĺžky, ktorý sa môže (ale nie nutne musí) líšiť od pôvodného readu na niektorých pozíciach.*

Fáza korekcie readov bude prebiehať tak, že ako prvé vygenerujeme sadu opravených readov. Potom skonštruujeme pole *diff* a nakoniec rozdelíme ready na dve skupiny, ako uvidíme neskôr.

Pole diff

Pole *diff* skonštruujeme tak, aby obsahovalo informácie o aplikovaných korekciách na jednotlivé ready. Formálne by sme ho definovali ako:

Definícia 3.1. *Množinu korekcií pre množinu readov R a korekčnú funkciu $corr_R$ definujeme ako*

$$diff = \{(corr_R(r), i, r[i]) \mid \forall r \in R\ \forall i : 0 \leq i < |r| : corr_R(r)[i] \neq r[i]\}$$

Algoritmus na konštrukciu poľa *diff* by vyzeral nasledovne:

```
1  diff = Array.new
2  correcter = Correcter.new
3  corrected_reads = correcter.correct_reads(R)
4
5  for (i = 0; i < R.length; i++) do
6    for (j = 0; j < R[i].length; j++) do
7      if R[i][j] != corrected_reads[i][j]
8        diff.push([i, j, R[j]])
9      end
10   end
11 end
```

Listing 3.3: Algoritmus konštrukcie poľa diff.

Algoritmus dostane na vstupe množinu readov R, ktoré opraví – trieda *Correcter* (riadok 2) predstavuje vhodné zapuzdrenie implementácie algoritmu opravujúceho ready – funkciu $corr_R$. Vo for cykle potom porovnáme každý znak každého readu s pôvodným, a ak nie sú zhodné, vložíme do poľa *diff* trojicu $[i, j, o]$, kde i predstavuje poradové číslo reťazca[4], j index ukazujúci na pozíciu v danom reade, kde sa pred korekciou nachádzal znak o. Na konci toto pole utriedime podľa prvej súradnice – to je dôležité z toho dôvodu, aby sme vedeli efektívne (pomocou binárneho vyhľadávania) vyhľadávať všetky korekcie pre daný read.

Rozdelenie readov

Definícia 3.2. *Uvažujme read r z množiny readov R, korekčnú funkciu $corr_R$ a parameter p[5]. Potom read r nazývame* kritický, *ak existujú také čísla i, j, že $0 \leq i < j < |r|$ a $corr_R(r)[i] \neq r[i] \wedge corr_R(r)[j] \neq r[j] \wedge j - i < p$.*

Poznámka 3.1. Nekritický read je taký read r z množiny readov R, ktorý nie je kritický.

Inak povedané, read je kritický, ak sú v opravenom reade dve korekcie príliš blízko seba – menej ako p.

Po tom, ako prebehne korekcia readov a konštrukcia poľa *diff*, rozdelíme ready na kritické – to sú pôvodné ready, ktorých korekcia by nám veľmi nepomohla – a opravené

[4]Mohli by sme použiť aj reťazec samotný, no týmto spôsobom vieme ušetriť nezanedbateľné množstvo pamäte.

[5]Dĺžka dotazu.

nekritické ready. Výstupom fázy korekcie readov teda budú dve sady readov a utriedené pole uchovávajúce informácie o aplikovaných korekciách.

3.3.2 Budovanie indexu

Hammingova vzdialenosť

Definícia 3.3. *Hammingova vzdialenosť dvoch reťazcov s a t rovnakej dĺžky l je definovaná nasledovne:*

$$edt_h(s,t) = |\{\ i \mid 0 \leq i < l\ \wedge s[i] \neq t[i]\ \}|$$

Inými slovami, pre dva rovnako dlhé reťazce je to počet pozícií, na ktorých sa líšia.

Príklad 3.4. $edt_h(ACCTGG, TCCTGC) = 2$

Superstring

Kým v doterajšom texte sme používali pojem superstring dosť voľne, na tomto mieste ho konečne riadne zadefinujeme.

Definícia 3.4. *Superstringom množiny reťazcov R rovnakej dĺžky l s parametrom $p \in \mathbb{N}$ označujeme taký reťazec S, pre ktorý platí, že každý reťazec $r \in R$ spĺňa aspoň jednu z nasledujúcich podmienok:*

(a) r je podreťazcom S

(b) $rev\_compl(r)$ je podreťazcom S

(c) existuje podreťazec s reťazca S s dĺžkou l taký, že

$$\forall i : 0 \leq i \leq l - p : edt_h(r[i, i+p-1], s[i, i+p-1]) \leq 1$$

(d) existuje podreťazec s reťazca S s dĺžkou l taký, že

$$\forall i : 0 \leq i \leq l - p : edt_h(rev\_compl(r)[i, i+p-1], s[i, i+p-1]) \leq 1$$

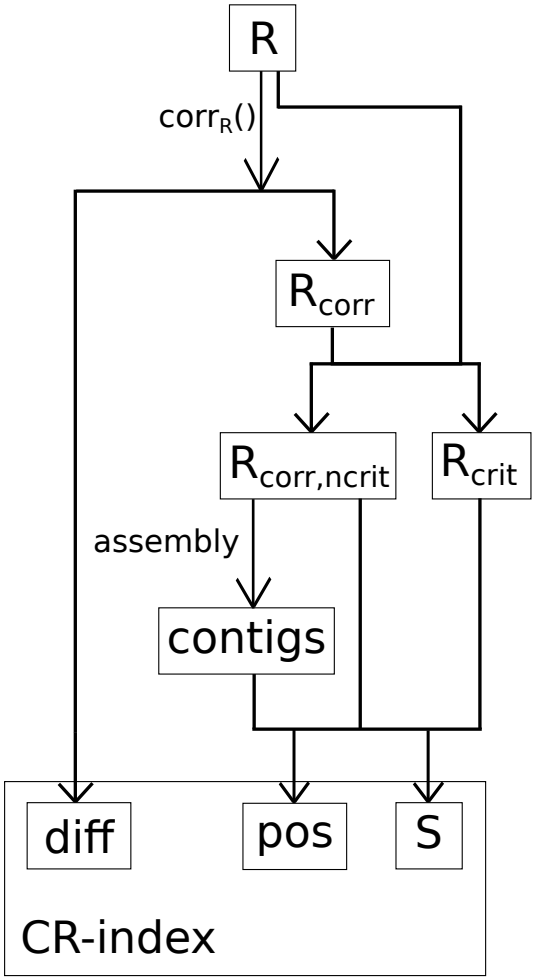
Povedané inak, superstringom množiny reťazcov R je taký reťazec, v ktorom pre každý reťazec r z R platí, že buď on alebo jeho reverzný komplement je podreťazcom superstringu alebo pre r alebo jeho reverzný komplement vieme nájsť podreťazec superstringu rovnakej dĺžky, ktorý sa od r (resp. jeho reverzného komplementu) „líši len tu a tam“ – to znamená, že indexy, na ktorých majú tieto dva reťazce rôzne znaky nie sú „príliš“ blízko seba, t.j. každé dva sú od seba vzdialené aspoň p.

Pre nás bude podstatné nájdenie čo najkratšieho superstringu pre danú sadu readov. Technicky vzaté, najkratší superstring je najkratšie spoločné nadslovo readov, no keďže jeho hľadanie je NP-ťažký problém, aplikujeme vhodnú heuristiku – zostavovanie genómu (*sequence assembly*) pomocou assemblera.

Index

Tak ako sme už naznačili, hlavná zmena bude spočívať v tom, že assembler dostane na vstup už opravené ready, no len nekritické. Tak ako v predchádzajúcej časti, výsledné kontigy zreťazíme a skonštruujeme nad nimi FM-index, pomocou ktorého identifikujeme tie ready, ktoré assembler dostal na vstup, ktoré nie sú podreťazcom zreťazenia kontigov a zreťazíme ich so zreťazením kontigov. Zároveň popritom konštruujeme pole *positions* s jemnou odlišnosťou oproti tomu, ako sme ho popísali v časti 3.1.2. Namiesto celého readu si totiž, kvôli úspore pamäte, budeme pamätať len identifikátor daného readu – jeho poradové číslo (vzhľadom na poradie vo vstupnej sade R). Na spätnú rekonštrukciu readu potom využijeme schopnosť FM-indexu rekonštruovať pôvodný text, nad ktorým bol konštruovaný – zaplatíme za to síce cenu v podobe dlhšieho času, ktorý bude potrebný na vykonanie dotazu, no úspora pamäte bude výrazná.

V poslednej fáze pripojíme k výslednému reťazcu neopravené kritické ready a konštrukcia superstringu je hotová. Z konštrukcie vyplýva, že takto skonštruovaný superstring spĺňa definíciu 3.4 – pre kritické ready zo vstupnej sady readov je splnená podmienka (a) z tejto definície a pre nekritické ready zo vstupnej sady buď podmienka (c) alebo (d) – tieto podmienky sú totiž postavené tak, že hovoria práve o tom, že v superstringu sa má nachádzať taký podreťazec, že sa od daného readu „veľmi nelíši“ a indexy, na ktorých sa tieto dva reťazce líšia majú byť od seba vzdialené aspoň p – a to nekritické ready spĺňajú. Nad výsledným superstringom skonštruujeme nový FM-index, ktorý spolu s poľom *diff* a (utriedeným) poľom *positions* tvoria celý CR-index.



Na obrázku je znázornená schéma konštrukcie CR-indexu. R označuje vstupnú sadu readov, R_{corr} je sada opravených readov – výstup korekčnej funkcie $corr_R$. Pole *diff* a sady readov $R_{corr,ncrit}$ (opravené, nekritické) a R_{crit} (pôvodné, kritické) tvoria výstup fázy korekcie readov popísanej v časti 3.3.1. Vo fáze budovania indexu (popísanej v časti 3.3.2) potom zo zreťazených kontigov, chýbajúcich nekritických opravených readov a kritických readov skonštruujeme superstring, ktorý spolu s poľami *diff* a *pos* (*positions*) tvoria samotný CR-index.

Obr. 3.1: Schéme konštrukcie CR-indexu

3.3.3 Dotazy

Kým v predchádzajúcej časti sme pri dotazovaní spravili len jednoduchý dotaz do FM-indexu a podľa výsledných indexov vyhľadali príslušné ready, tým, že superstring obsahuje opravené ready[6] je momentálna situácia o čosi komplikovanejšia.

Proces dotazu rozdelíme na dve fázy – v prvej fáze sa FM-indexu spýtame na výskyty vzorky p, k nim zrekonštruujeme prislúchajúce ready a vylúčime prípadné falošné detekcie (*false positives*) – mohla totiž nastať situácia, že opravený read obsahuje výskyt vzorky p, takže ho FM-index nájde, no v pôvodnom reade sa výskyt p nenachádza.

V druhej fáze vykonáme dotazy pre všetky reťazce s Hammingovou vzdialenosťou 1 od vzorky p (a opäť zrekonštruujeme ready a overíme). Táto druhá fáza je potrebná kvôli tomu, aby sme niektoré ready, ktoré p obsahujú nevynechali z výsledku. Mohlo totiž nastať to, že korekcia readu prebehla akurát na mieste, kde tento read obsahuje vzorku p a ak tento read nie je kritický, tak v superstringu sa nachádza akurát s týmto

[6] a originálne kritické ready

modifikovaným výskytom vzorky p. Na druhej strane, ak pri nejakom reade nastali také korekcie, že boli bližšie ako p, t.j. tak sa v superstringu nachádza jeho originál[7], takže nám stačí Hammingova vzdialenosť 1. Algoritmus na vykonanie dotazu teda vyzerá nasledovne:

```
1  def locate(p)
2    retval = locate_with_check(p, p)
3
4    foreach p2 : strings_with_hd_1(p) do
5      foreach i : locate_with_check(p2, p) do
6        retval.push(i)
7      end
8    end
9
10   return retval
11 end
12
13 def locate_with_check(p, p_check)
14   retval = Array.new
15   indexes = fm_index.locate(p)
16
17   foreach i : indexes do
18     foreach pos : positions.bound([i + k - l, -1, 0],
19                                   [i, INT_MAX, 1]) do
20       read_id = pos[1]
21       diffs = diff.bound([read_id, -1, 'A'],
22                          [read_id, INT_MAX, 'A'])
23       if pos[2]
24         p2 = apply_corrections(diffs, rev_compl(p))
25         if p2 == rev_compl(p_check)
26           retval.push(read_id)
27         end
28       else
29         p2 = apply_corrections(diffs, p)
30         if p2 == p_check
31           retval.push(read_id)
32         end
```

[7] Keďže je kritický.

```
33        end
34      end
35    end
36
37    return retval
38  end
```

Listing 3.4: Algoritmus dotazu *locate* CR-indexu nad readmi s chybami.

Funkcia *locate*, ako sme už skôr spomenuli, najprv vykoná dotaz pre vzorku p (riadok 2) a potom vykonáva dotazy pre všetky vzorky s Hammingovou editačnou vzdialenosťou 1 od vzorky p – riadok 5 (predpokladajme ešte, že máme k dispozícii funkciu *strings_with_hd_1(s)*, ktorá pre daný reťazec vráti pole všetkých reťazcov s Hammingovou vzdialenosťou 1 od reťazca s).

Zaujímavejšia je funkcia *locate_with_check* (riadok 13). Tá vyhľadá vzorku p, pre každý jej výskyt nájde prislúchajúce ready (riadok 18) – kvôli zjednodušeniu zápisu predpokladáme, že máme k dispozícii metódu *bound*, ktorá vracia podpole ohraničené argumentmi, čiže je to len skrátený zápis riadkov 6 – 12 z príkladu 3.2. Ďalej pre každý read nájdeme prislúchajúce korekcie v poli *diff* (riadky 21 a 22). Funkcia *apply_corrections*[8] potom aplikuje relevantné[9] korekcie na nájdený výskyt p, aby sme zistili, ako tento nájdený výskyt p vyzeral v pôvodnom, neopravenom reade – ak je rovnaký ako *p_check*, tak sme úspešne našli výskyt a pridáme daný read do poľa výsledkov.

Celé fungovanie bude jasnejšie na príklade:

Príklad 3.5. Uvažujme $k = 4$, $l = 8$, nasledovný superstring[10] (nech index jeho prvého znaku je 100):

$$S = \ldots ACCTAGCCTGCTTTAGCTTTAG \ldots$$

Ďalej nech polia *positions* a *diff* vyzerajú nasledovne:

[8] Implementáciu tejto funkcie sme zámerne vynechali, nakoľko nie je veľmi myšlienkovo zaujímavá a prevládajú v nej skôr technické detaily týkajúce sa prerátavania indexov.

[9] Nie je potrebné rekonštruovať kompletný originálny read, stačí nám tá časť, ktorá obsahuje p.

[10] Respektíve jeho časť.

i	read_id	rev_compl?
...	...	...
100	966	0
102	256	0
...	...	...
110	452	0
...	...	...
113	812	0
...	...	...

read_id	i	orig
...	...	...
254	7	C
256	0	A
256	7	C
257	1	T
...	...	...
812	4	C
...	...	...
966	6	G
...	...	...

Ilustrujme teraz dotaz pre vzorku $p = AGCC$. FM-index nájde výskyt tejto vzorky na pozícii 104:

$$S = \ldots ACCT\mathbf{AGCC}TGCTTTAGCTTTAG\ldots$$

V tabuľke *positions* nájdeme relevantné záznamy – vidíme, že ready číslo 966 a 254 obsahujú tento výskyt p. Následne pre tieto ready vyhľadáme záznamy o korekciách v tabuľke *diff*. Vidíme, že pri reade 256 nastali dve korekcie – znaky na indexoch 0 a 7 sa zmenili na A resp. C – keď ale tieto indexy prerátame na indexy v rámci superstringu (102 resp. 109) tak vidíme, že tieto nezasahujú do tohto výskytu p (ten sa nachádza v superstringu na pozíciach 104 – 108). Z toho vyplýva, že read číslo 256 tento výskyt p obsahuje.

Ďalej spracujeme read číslo 966 začínajúci na indexe 100 v rámci superstringu. V tomto prípade korekcia nastala na indexe 6 vzhľadom na tento read, čiže na indexe 106 v rámci superstringu – to už zasahuje do výskytu vzorky p, z ktorej tým dostávame $p' = AG\mathbf{G}C$, čo znamená, že read číslo 966 výskyt p neobsahuje, namiesto neho obsahuje podreťazec $AGGC$, ktorý sa vo fáze korekcie readov zmenil na $AGCC$ a preto sme tento výskyt našli v superstringu.

Nasleduje druhá fáza dotazu, kde spravíme dotaz pre všetky reťazce s Hammingovou vzdialenosťou 1 od p. Pre účely príkladu si vystačíme s jednou vzorkou, napríklad $p_{d(1)} = AGCT$. FM-index vráti ako výskyt tejto vzorky index 114 v superstringu:

$$S = \ldots ACCTAGCCTGCTTT\mathbf{AGCT}TTAG\ldots$$

Prvým nájdeným readom, ktorý obsahuje $AGCT$ je read číslo 452 – v tabuľke *diff*

pre tento read ale nemáme žiadne záznamy, čo znamená, že pri ňom neprebehli žiadne korekcie. Obsahuje síce výskyt modifikovanej vzorky – $AGCT$, no nie pôvodnú vzorku p a teda sa do výsledku nedostane. Druhý read, ktorý obsahuje $AGCT$ je read číslo 812. Preň máme v tabuľke *diff* záznam o korekcii na indexe 4. Keďže tento read začína v superstringu na indexe 113, tak táto korekcia sa dotkne aj výskytu hľadanej vzorky – $AGCT$ sa upraví na $AGCC$, čo sa rovná pôvodne hľadanej vzorke p a preto read číslo 452 tiež zahrnieme do výsledku. Čo vlastne nastalo je to, že read číslo 812 pôvodne obsahoval ako podreťazec p, no pri korekčnej fáze bol tento read zmenený akurát na mieste výskytu p. Spätným hľadaním pomocou poľa *diff* sme to ale odhalili a úspešne ho zahrnuli do výsledku.

Poznámka 3.2. V predchádzajúcom príklade sme zámerne uvažovali len ready, ktoré sa v superstringu nachádzajú priamo samé a nie ich reverzný komplement. Princíp hľadania by sa nezmenil (postupovali by sme podľa algoritmu uvedenom v listingu 3.4), no príklad by bol zbytočne komplikovaný a technický.

Poznámka 3.3. Tak ako sme už uviedli v časti 3.1.3, tak „pre splnenie poslednej podmienky z časti 2.1 by sme pre dotaz na vzorku p mali zavolať metódu *locate* aj pre vstup $rev\_compl(p)$ a výstupy pre obe volania spojiť do jedného poľa a vrátiť ako výstup to“.

3.3.4 Možné zlepšenia

Naša implementácia CR-indexu ako riešenia *problému indexovania readov* je síce pamäťovo výrazne efektívnejšia než implementácia pomocou hash mapy alebo použite knižnice GkArray, no stále je tu priestor na zlepšenie.

V prvom rade by sa dali polia *diff* a *positions* reprezentovať efektívnejšie pomocou tzv. *úsporných dátových štruktúr* (*succinct data structures*) [GBMP14].

Ďalším, už podstatne menej triviálnym zlepšením by bolo použitie *Bloom filtra*, v druhej fáze dotazu, keď sa pýtame CR-indexu na vzorky s Hammingovou vzdialenosťou 1 od pôvodnej vzorky. Bloom filter je pamäťovo efektívna pravdepodobnostná dátová štruktúra, ktorá sa používa na odpovedanie na otázku, či sa daný prvok nachádza v danej množine. Bloom filter je navrhnutý tak, že sa môže stať, že odpovie, že sa daný prvok v množine nachádza, aj keď sa tam nenachádza (*false positive*), no nikdy nie naopak. Pri konštrukcii CR-indexu, konkrétne pri konštrukcii poľa *positions* by sme pre každý spracovávaný read vygenerovali všetky jeho podreťazce dĺžky p a tie vložili do Bloom filtra. Pri druhej fáze dotazu by sme potom vedeli vynechať niektoré dotazy na FM-index tým, že by sme sa najprv Bloom filtra spýtali, či má zaindexovaný

daný reťazec. Týmto spôsobom by sme síce mierne predĺžili čas potrebný na konštrukciu CR-indexu a použili isté množstvo pamäte navyše, no na druhej strane by sme nezanedbateľne zrýchlili operáciu *locate*.

Kapitola 4

Implementácia

V tejto kapitole popíšeme našu implementáciu CR-indexu v jazyku `C++`, použité knižnice, a jej štruktúru a rozhrania.

4.1 Použité knižnice

Ako programovací jazyk sme zvolili `C++`, konkrétne verziu `C++11`. Zdrojový kód by mal byť (po doinštalovaní potrebných závislostí) skompilovateľný kompilátorom `gcc` verzie 4.9.1 a vyššie. Pri implementácii sme využili nasledujúce knižnice resp. programy:

Boost

Boost je rozsiahla sada univerzálnych knižníc pre programovací jazyk `C++`. My sme z nej využili podknižnice `libboost-filesystem` a `libboost-system` na prácu so súborovým systémom a procesmi.

SGA

SGA (*String Graph Assembler*) [SD11] je implementácia *de novo genome* assemblera založeného na koncepte grafových reťazcov v jazyku `C++` a je dostupná na stránke `https://github.com/jts/sga`. Tento assembler sme využili pri implementácii CR-indexu hneď na niekoľkých miestach:

- V časti 3.3.1 – Korekcia readov, sme realizovali korekčnú funkciu $corr_R$ pomocou podprogramu SGA `sga correct`. Ten očakáva na vstupe okrem sady readov vo `fastq` formáte aj index tejto sady, ktorý sme vybudovali pomocou podprogramu `sga index`.

- Ďalej v časti 3.3.2 – Budovanie indexu sme pomocou SGA konštruovali kontigy. Najprv sme pomocou `sga index` vybudovali index readov, potom pomocou `sga overlap` našli prekryvy readov a nakoniec cez `sga assemble` skonštruovali kontigy.

SDSL

SDSL (*Succinct Data Structure Library*) [GBMP14] je sada knižníc obsahujúca implementáciu *úsporných dátových štruktúr* pre programovací jazyk `C++`. Zahŕňa množstvo rôznych implementácií napríklad bitových vektorov, celočíselných vektorov, wavelet stromov, komprimovaných sufixových polí a podobne. My sme túto knižnicu využili na implementáciu FM-indexu, ktorý používame v CR-indexe hneď dvakrát – prvýkrát v konštrukčnej fáze, keď v zreťazených kontigoch hľadáme chýbajúce ready a druhýkrát konštruujeme FM-index nad superstringom a potom ho využívame pri dotazovaní.

4.2 Inštalácia

Zdrojový kód našej implementácie je verejne dostupný na Github-e na URL `https://github.com/kuboj/CR-index` pod licenciou MIT. Pred vybudovaním zdrojových kódov CR-indexu je najprv potrebné mať nainštalované všetky závislosti spomínané v predchádzajúcej časti. `Boost`, `PStreams`, `libboost-filesystem` a `libboost-system` zvyknú byť široko dostupné pre Unixové operačné systémy vo forme balíčkov pre daný systém. Po naklonovaní repozitára lokálne stačí spustiť príkaz `make` pre vybudovanie modulov alebo `make examples` resp. `make benchmarks` pre vybudovanie príkladov respektíve testovacích programov.

4.3 Vonkajšia štruktúra

Súborová štruktúra

`benchmark/`	obsahuje dva testy – `construct.cpp` a `query.cpp`
`bin/`	táto zložka po vybudovaní obsahuje spustiteľné súbory
`build/`	obsahuje vybudované objektové súbory
`examples/`	v tejto zložke sa nachádzajú príklady
`include/`	obsahuje hlavičkové súbory (*header files*)
`src/`	obsahuje samotné zdrojové kódy knižnice
`tools/`	zoskupuje pomocné nástroje
`LICENSE`	súbor obsahujúci MIT licenciu
`Makefile`	obsahuje pravidlá pre vybudovanie knižnice pomocou nástroja `make`
`README.md`	stručné inštrukcie pre inštaláciu a používanie

benchmark/construct.cpp

Úlohou tohto testu je odmerať množstvo spotrebovanej pamäte (a čas potrebný pre konštrukciu) pre jednotlivé implementácie štruktúr indexujúcich ready. Jeho rozhranie príkazového riadku vyzerá nasledovne:

```
1 $ construct --help
2 Construct index
3 Usage:
4   construct <index_type> <filename> <read_length> <query_length>
5 index_type: cr, hash, gk
6 Example:
7   ./bin/construct cr bacteria.fastq 100 13
```

Listing 4.1: Rozhranie príkazového riadku test construct.cpp

Na vstupe očakáva typ indexu, ktorým bude indexovať vstupnú sadu readov – `cr` označuje našu implementáciu CR-indexu, `hash` implementáciu pomocou hash mapy (mierne vylepšenú oproti časti 2.2) a `gk` označuje konštrukciu indexu pomocou knižnice GkArray (ktorá je popísaná v časti 2.3). Parameter `filename` predstavuje cestu k súboru so sadou readov vo formáte FASTQ, parameter `read_length` označuje dĺžku readu a posledný parameter, `query_length` dĺžku dotazu (ktorú sme v predchádzajúcom texte označovali ako k).

Ukážkový výstup tohto testu je nasledovný:

```
$ ./construct cr ../genomes/bacteria1M_10E.fastq 100 13
index_type: cr
read_filename: ../genomes/bacteria1M_10E.fastq
read_length: 100
query_length: 13
Total reads size: 100000000
Superstring size:  9456094
Compress ratio: 10.575191
Construction took 252.935495s
Referenced memory: 23536kB
OK.
```

Listing 4.2: Ukážkový výstup testu construct.cpp

Meranie spotrebovaného množstva pamäte prebieha na konci konštrukcie – snahou je teda zmerať, koľko pamäte zaberá samotný index a nie maximum spotrebovanej pamäte počas konštrukcie[1]. Toto meranie je (momentálne) implementované len pre Unixové operačné systémy a je realizované pomocou analýzy mapovania pamäti procesu, ktoré čítame zo súboru `/proc/<pid>/smaps`.

benchmark/query.cpp

Tento najprv skonštruuje daný typ indexu a potom meria čas potrebný na odpovedanie na všetky dotazy zo vstupného súboru. Rozhranie príkazového riadku je nasledovné:

```
$ ./query
Construct index
Usage: ./query <index_type> <reads_filename> <queries_filename>\
  <read_length> <query_length> <query_type>
index_type: cr, hash, gk
query_type: index, read
Example:
  ./query cr bacteria.fastq queries.data 101 13 index
```

Listing 4.3: Rozhranie príkazového riadku test query.cpp

Parametre `index_type`, `reads_filename`, `read_length`, `query_length` sú rovnaké ako v predchádzajúcom teste. Parameter `queries_filename` označuje cestu

[1] Aj keď aj to je často sledovaný parameter.

ku súboru obsahujúcemu sadu dotazov, každý dĺžky `query_length`, oddelených znakom nového riadku. Parameter `query_type` obsahuje typ dotazu, ktorý vykonávame – v prípade, že sa pýtame len na indexy readov, ktoré danú vzorku obsahujú je tento rovný *index*, ak požadujeme pri dotaze vrátenie celých readov ako reťazcov, tak tento parameter musí mať hodnotu `read`.

Ukážkový výstup tohto testu je nasledovný:

```
$ ./query cr ../genomes/bacteria1M_10E.fastq queries 100 13 index
index_type: cr
read_filename: ../genomes/bacteria1M_10E.fastq
queries_filename: queries
read_length: 100
query_length: 13
query_type: index
number of queries: 10000
Constructing index ...
Total reads size: 100000000
Superstring size:  9456094
Compress ratio: 10.575191
Querying ...
Querying took 2.399451s
OK.
```

Listing 4.4: Ukážkový výstup testu query.cpp

4.4 Vnútorná štruktúra

Samotné zdrojové kódy CR-indexu sa nachádzajú v zložkách `include/` (hlavičkové súbory) a `src/` (súbory implementácie). Zložka `src/` obsahuje súbory `util.cpp` (čo je súbor pomocných funkcií na prácu s reťazcami, spúšťanie podprocesov a podobne), `hash_index.cpp` (predstavuje implementáciu pomocou hash mapy), `fm_wrapper.cpp` (trieda vhodne zapuzdrujúca implementáciu FM-indexu v knižnici SDSL) a súbor `cr_index.cpp`, ktorý implementuje triedu CRIndex.

Trieda CRIndex

Verejné rozhranie tejto triedy vyzerá nasledovne:

```
1  class CRIndex {
2    public:
3      static const bool DEFAULT_VERBOSITY;
4      static const int DEFAULT_READ_LENGTH;
5      static bool verbose;
6
7      CRIndex(string path, int read_length = DEFAULT_READ_LENGTH,
8                  bool verbose = DEFAULT_VERBOSITY);
9      CRIndex(string superstring, vector<t_pos> positions,
10                 vector<t_diff> diff,
11                 int read_length = DEFAULT_READ_LENGTH,
12                 bool verbose = DEFAULT_VERBOSITY);
13     vector<int> find_indexes(const string& s);
14     vector<string> find_reads(const string& s);
15     ~CRIndex();
16
17     static tuple<string, vector<t_pos>, vector<t_diff>>
18                       preprocess(string path,
19                           bool verbose = DEFAULT_VERBOSITY);
20     ...
```

Listing 4.5: Verejné rozhranie triedy CRIndex

Typy `t_pos` a `t_diff` označujú typ prvkov polí `pos` resp. `diff` (a sú definované v príslušnom hlavičkovom súbore).

Prvý konštruktor očakáva na vstupe reťazec s cestou ku súboru so sadou readov vo formáte FASTQ, dĺžku readu a logickú premennú, ktorá zapína obsiahlejšie výpisy. Tento konštruktor kompletne vykoná konštrukciu celého CR-indexu.

Statická metóda `preprocess` spracuje vstupnú sadu readov a vráti superstring a polia `pos` a `diff`. Túto metódu je možné použiť s druhým konštruktorom – ak by sme potrebovali tieto medzivýsledky uložiť alebo nejak analyzovať.

Dotazy sa vykonávajú pomocou metód `find_reads` a `find_indexes`, ktoré pre reťazec na vstupe vrátia ready, resp. indexy readov, ktoré túto vzorku obsahujú[2].

Minimalistický príklad použitia by teda vyzeral nasledovne:

[2]Alebo jej reverzný komplement.

```
...
#include "cr_index.hpp"

CRIndex cr = CRIndex("/tmp/staphylococcus.fastq", 101);
for (auto s : cr.find_reads("ACTGGGTCCACCCA")) {
  cout << s << endl;
}
```

Listing 4.6: Príklad použitia triedy CRIndex

Kapitola 5

Výsledky

V poslednej časti práce prezentujeme praktické výsledky našej implementácie CR-indexu.

5.1 Testovacie prostredie

Všetky testy prebiehali pod 64-bitovou verziou operačného systému Linux verzie 3.16.0-34. Testy boli kompilované kompilátorom `gcc` verzie 4.9.1 s prepínačmi `-std=c++11` a `-O3` a spustené na hardvéri s CPU `Intel(R) Core(TM) i7-4910MQ CPU @ 2.90GHz` a 16GB RAM. Na meranie spotreby pamäte a rýchlosti odpovedania na dotazy sme použili testy `benchmark/construct.cpp` a `benchmark/query.cpp` popísané v časti 4.3.

5.2 Testy

5.2.1 CR-index vs GkArray - pamäť

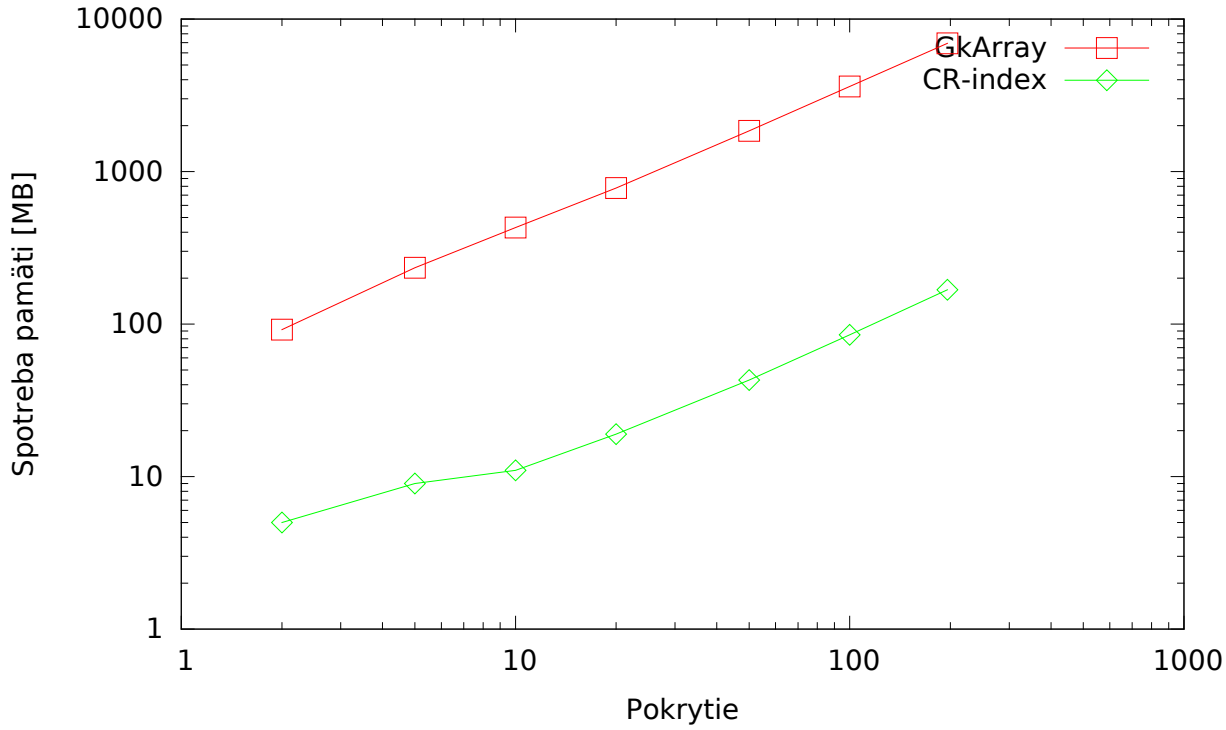
V tomto teste sme porovnávali spotrebu pamäte CR-indexu oproti knižnici GkArray. Pre testovanie sme použili sadu readov spracovávanú v [DFPB13][1]. Sekvenovaný genóm patrí baktérii *Escherichia Coli* a má dĺžku 4639675 báz. Táto sada obsahuje 6028744[2] readov dĺžky 151 (z čoho vyplýva, že pokrytie sa pohybuje na úrovni 196.2×) a miera chýb predstavuje 0.75%. Dĺžku dotazu sme zvolili $k = 13$.

[1] Táto sada je verejne dostupná na internete na linke `ftp://webdata:webdata@ussd-ftp.illumina.com/Data/SequencingRuns/MG1655/MiSeq_Ecoli_MG1655_110721_R1.fastq`

[2] Z pôvodnej sady sme vynechali tie ready, ktoré obsahovali neznámu bázu (N), ich podiel bol zanedbateľný.

Z tejto základnej sady readov sme potom *samplovaním* vyrobili ďalšie, pomocou ktorých sme simulovali pokrytie na úrovni 2×, 5×, 10×, 20×, 50× a 100×.

Namerané výsledky možno vidieť v nasledujúcom grafe:

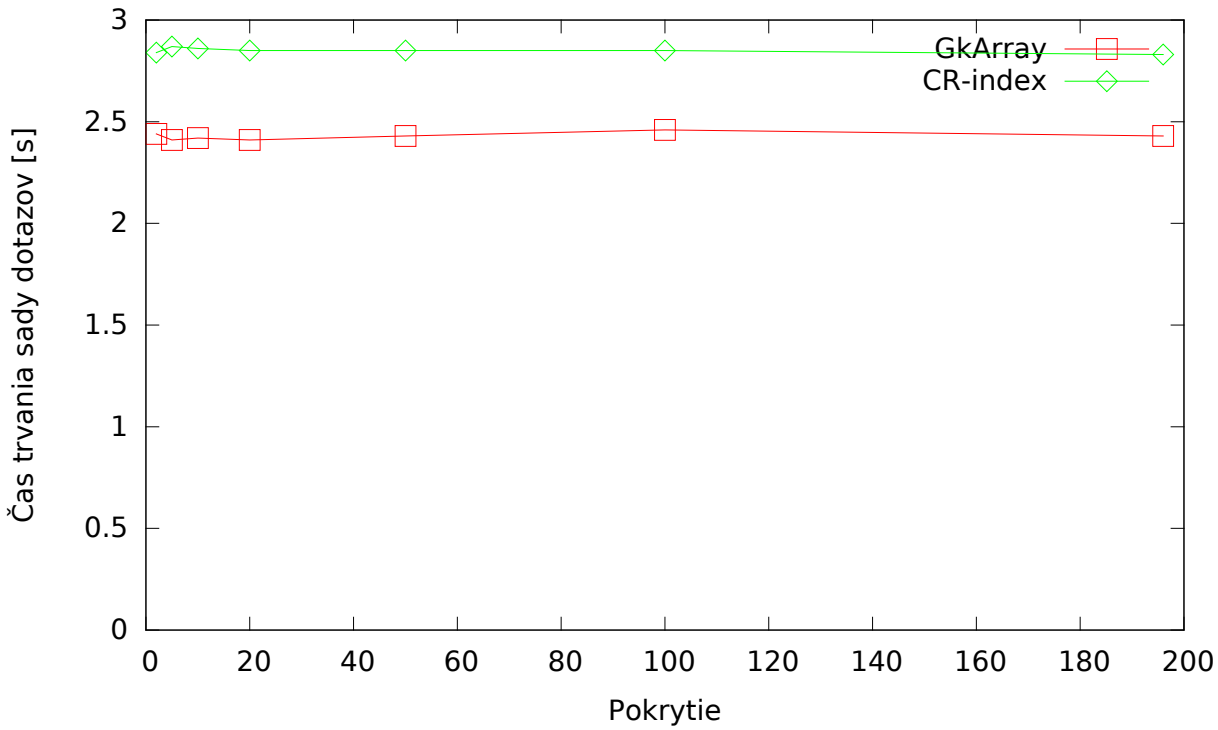


Obr. 5.1: Porovnanie spotreby pamäti CR-indexu a GkArray pre rôzne pokrytia.

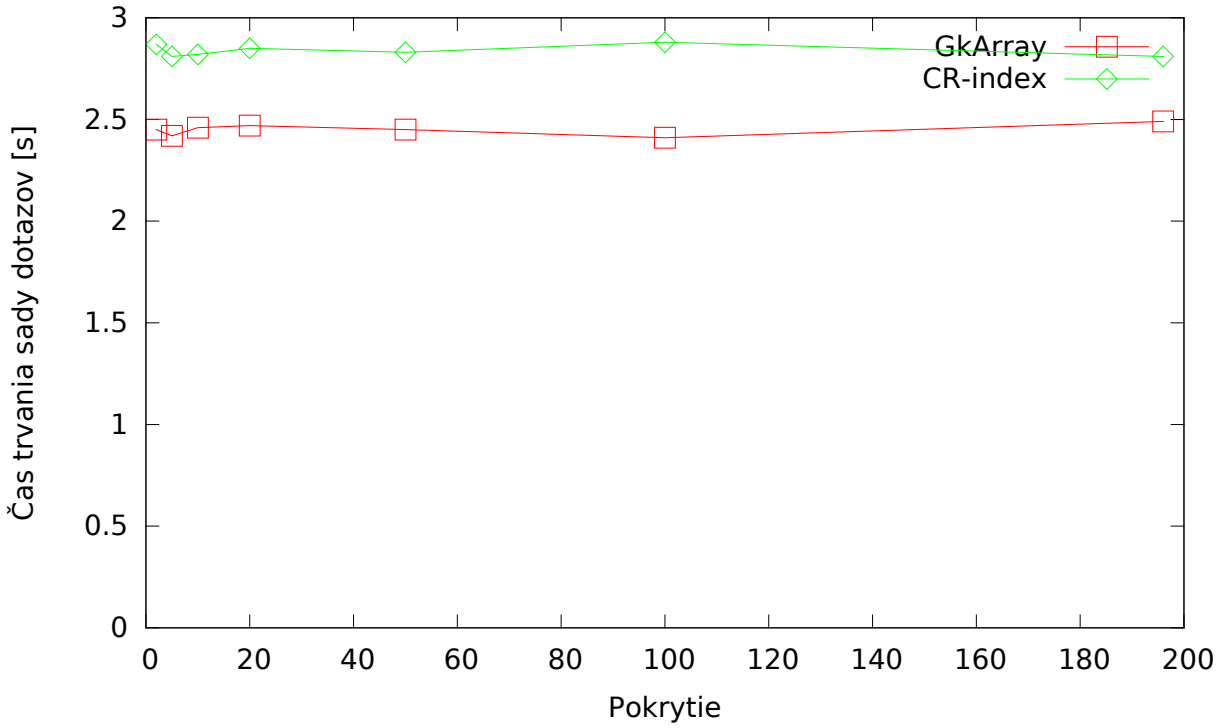
5.2.2 CR-index vs GkArray - čas dotazu

Tento test porovnáva rýchlosť odpovedania na dotazy pre CR-index a knižnicu GkArray. Sady readov sme použili rovnaké ako v predchádzajúcom teste. Sady dotazov (pre $k = 13$) sme použili dvoch typov – prvá sada bola plne náhodne generovaná a druhá bola generovaná ako náhodné podreťazce dĺžky k náhodných readov z danej sady readov. Počet dotazov pre každú sadu činil 100 miliónov (z toho dôvodu, aby sme čo najviac obmedzili odchýlky a zároveň aby sa táto sada aj s daným indexom pohodlne zmestili do operačnej pamäte počítača).

Tabuľka 5.2 ilustruje namerané hodnoty pre sadu 100 miliónov náhodných dotazov a tabuľka 5.3 pre sadu 100 miliónov generovaných readov.



Obr. 5.2: Porovnanie času potrebného pre vykonanie sady náhodných dotazov pre CR-indexu a GkArray pre rôzne pokrytia.

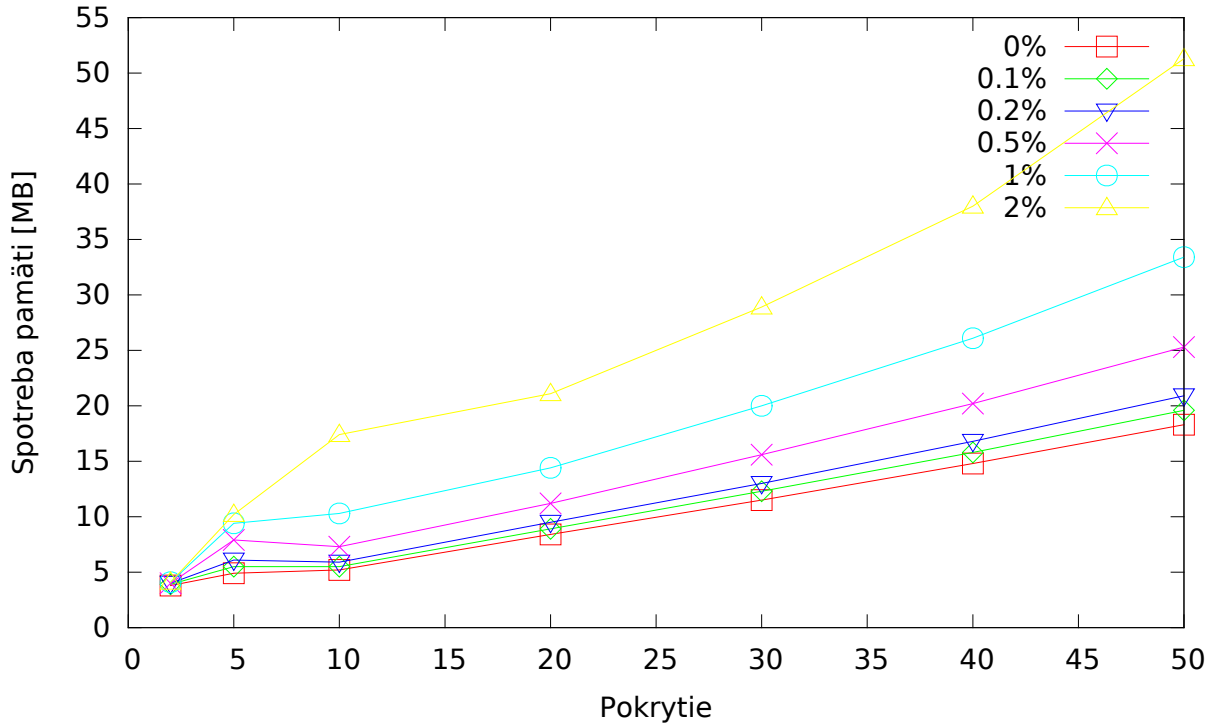


Obr. 5.3: Porovnanie času potrebného pre vykonanie sady nenáhodných dotazov pre CR-indexu a GkArray pre rôzne pokrytia.

5.2.3 Pamäť vs. pokrytie

V tomto teste sme merali závislosť spotreby pamäte na pokrytí pre rôznu chybovosť readov. Sady readov sme generovali z genómu baktérie *Staphylococcus aureus*[3][4]. Tento genóm má dĺžku 2872915 báz, dĺžku readu l sme zvolili 100 a dĺžku dotazu $k = 13$.

Podľa týchto parametrov sme dorátali, že počet readov v sadách pre jednotlivé pokrytia je: 57458 readov pre pokrytie 2×, 143645 readov pre pokrytie 5×, 287291 readov pre pokrytie 10×, 574583 readov pre pokrytie 20× a 1436457 readov pre pokrytie 50×.



Obr. 5.4: Závislosť spotreby pamäte na pokrytí pre rôznu chybovosť vstupnej sady readov.

[3]Dostupné na internete na stránke `http://gage.cbcb.umd.edu/data/index.html`

[4]Z genómu na stránke sme odstránili genómy plazmidov a použili len genóm samotnej baktérie.

5.3 Zhrnutie a interpretácia výsledkov

Z výsledkov vykonaných testov môžeme skonštatovať, že úlohu naimplementovať pamäťovo efektívnu dátovú štruktúru indexujúcu ready sme úspešne splnili. CR-index vykazuje oproti knižnici GkArray rádovo nižšiu spotrebu pamäte pri zachovaní podobnej rýchlosti odpovedania na dotazy.

Ďalej vidíme, že čas odpovedania na dotaz nezávisí od pokrytia readmi ani v prípade CR-indexu ani pri knižnici GkArray a tiež, že to, že odpovedanie na náhodne generované dotazy je rovnako rýchle ako odpovedanie na dotazy, ktoré zaručene vrátia nejaký výsledok (generovaná sada).

Možnosti zlepšenia vidíme, ako sme spomínali už v časti 3.3.4, napríklad v implementácii bloom filtra, ktorým by sme vedeli zlepšiť čas potrebný na vykonanie dotazu a tým by sme sa priblížili ku času, ktorý vykazuje knižnica GkArray. Ďalšie možné zlepšenie je efektívnejšie ukladať polia *pos* a *diff* – pri vyšších pokrytiach práve tieto dve polia zaberajú väčšinu pamäte – superstring totiž v pamäti uložený celý nemáme, FM-index drží iba vhodne komprimovanú verziu.

Záver

V tejto práci sme sa venovali návrhu a implementácii dátovej štruktúry *CR-index*, ktorá rieši *problém indexovania readov* tak, ako sme ho zadefinovali – ako efektívne indexovať vstupnú sadu readov a vedieť v nich vyhľadávať ready, ktoré obsahujú daný podreťazec.

Túto dátovú štruktúru sme naimplementovali ako knižnicu pre jazyk `C++` a uverejnili na stránke `https://github.com/kuboj/CR-index/`. Pri meraniach jej vlastností sme potvrdili predpoklad pamäťovej efektivity, ktorý vyplýval z jej návrhu a ukázali sme, že naša implementácia je výrazne efektívnejšia ako existujúce riešenia.

V našej implementácii je však stále priestor na zlepšenia – ako napríklad implementácia Bloom filtra, ktorý by znížil čas potrebný na vykonanie dotazu alebo pamäťovo efektívnejšia reprezentácia polí *diff* a *positions*.

Literatúra

[BB13] Broňa Brejová, *Skriptá k predmetu vyhľadávanie v texte*, FMFI UK, 2013.

[BV11] Broňa Brejová a Tomáš Vinař, *Metódy v bioinformatike*, FMFI UK, 2011.

[BW94] Michael Burrows a David J Wheeler, *A block-sorting lossless data compression algorithm*, Technical Report 124, 1994.

[FM00] Paolo Ferragina a Giovanni Manzini, *Opportunistic data structures with applications*, Proceedings of the 41st Annual Symposium on Foundations of Computer Science, strany 390–398, 2000.

[PTW01] Pavel A. Pevzner, Haixu Tang a Michael S. Waterman, *An Eulerian path approach to DNA fragment assembly*, Proc Natl Acad Sci USA 2001.

[MM90] Udi Manber a Gene Myers, *Suffix arrays: a new method for on-line string searches*, First Annual ACM-SIAM Symposium on Discrete Algorithms, strany 319–327, 1990.

[AKO04] Mohamed Ibrahim Abouelhodaa, Stefan Kurtzb a Enno Ohlebuscha, *Replacing suffix trees with enhanced suffix arrays*, Journal of Discrete Algorithms 2, 2004.

[NZC09] Ge Nong, Sen Zhang a Wai Hong Chan, *Linear Suffix Array Construction by Almost Pure Induced-Sorting*, 2009 Data Compression Conference, strany 193–202, 2009.

[KS03] Juha Kärkkäinen a Peter Sanders, *Simple Linear Work Suffix Array Construction*, Automata, Languages and Programming. Lecture Notes in Computer Science 2719, strana 943, 2003.

[GGV03] Roberto Grossi, Ankur Gupta a Jeffrey Scott Vitter, *High-order entropy-compressed text indexes*, Proceedings of the 14th Annual SIAM/ACM Symposium on Discrete Algorithms, strany 841–850, 2003.

[JS12] Jochen Singer, *A Wavelet Tree Based FM-Index for Biological Sequences in SeqAn*, Master thesis, Freie Universität Berlin, 2012.

[NP11] Nicolas Phillipe, Mikaël Salson, Thierry Lecroq, Martine Léonard, Thérèse Commes a Eric Rivals, *Querying large read collections in main memory: a versatile data structure*, BMC Bioinformatics 2011, 12:242, 2011.

[SD11] Jared T. Simpson a Richard Durbin, *Efficient de novo assembly of large genomes using compressed data structures*, Genome Research 2012, 22: 549–556, 2011.

[GBMP14] Simon Gog, Timo Beller, Alistair Moffat a Matthias Petri, *From Theory to Practice: Plug and Play with Succinct Data Structures*, 13th International Symposium on Experimental Algorithms, strany 326–337, 2014.

[DFPB13] Viraj Deshpande, Eric DK Fung, Son Pham a Vineet Bafna, *Cerulean: A hybrid assembly using high throughput short and long reads*, 13th Workshop on Algorithms in Bioinformatics, 2013.

[ZS00] Z. Sweedyk *A 2.5-approximation Algorithm for Shortest Superstring*, SIAM Journal on Computing 29, strany 954–986, 2000.

[GJ02] Michael Garey a David S. Johnson *Computers and intractability*, W.H. Freeman, 2002.

[GGV03] Roberto Grossi, Ankur Gupta a Jeffrey S. Vitter, *High-Order Entropy-Compressed Text Indexes*, 14th Annual ACM-SIAM Symposium on Discrete Algorithms, strany 841–850, 2003.

[BBV14] Vladimír Boža, Broňa Brejová a Tomáš Vinař, *GAML: Genome Assembly by Maximum Likelihood*, Algorithms in Bioinformatics, strany 122–134, 2014.

[PSCR13] Nicolas Philippe, Mikaël Salson, Thérèse Commes a Eric Rivals, *CRAC: an integrated approach to the analysis of RNA-seq reads*, Genome biology, 2013.

[KSS10] David R. Kelley, Michael C. Schatz, Steven L. Salzberg et al., *Quake: quality-aware detection and correction of sequencing errors*, Genome Biology, 2010

[IFI11] Lucian Ilie, Farideh Fazayeli, a Silvana Ilie, *HiTEC: accurate error correction in high-throughput sequencing data*, Bioinformatics, strany 295–302, 2011.