Runminimierung einer Multi-String BWT

Uwe Baier

November 5, 2019

1 Überblick

In diesem Projekt soll die Zahl der Runs einer Multi-String BWT minimiert werden. Das typische Einsatzgebiet dieser Minimierung ist Datenkompression, da bei einer lauflängenkodierten BWT die Zahl der Runs eine entscheidende Bedeutung für die Komprimierbarkeit hat.

Die grundlegende Idee hierzu ist, dass Buchstaben in der BWT, die vor gleichen lexikographisch geordneten Suffixen stehen, in beliebiger Reihenfolge permutiert werden können. Ziel des Projektes soll es sein, die in diesem Konzept beschriebenen Algorithmen zu implementieren und anhand von später bereitgestellten experimentellen Daten die Wirksamkeit der Runminimierung zu untersuchen. Die zugrundeliegende Programmiersprache bildet hierbei C++, benötigte Datenstrukturen werden in Form der `sdsl-lite` [5] Library bereitgestellt.

2 Definitionen

In diesem Konzept wird jedes Interval $[i, j]$ oder $[i, j)$ als ein Interval über den natürlichen Zahlen verstanden, jeder Logarithmus hat Basis 2, und Indizes starten immer von 1. Die meisten der folgenden Definitionen stammen aus [**?**].

Sei Σ eine total geordnete Menge (Alphabet) von Buchstaben. Ein String S der Länge n über dem Alphabet Σ ist eine endliche Sequenz von n Buchstaben aus dem Alphabet Σ. Wir nennen S nullterminiert falls S mit dem kleinsten Buchstaben \$ endet, der nur am Ende von S vorkommt. Der leere String mit Länge 0 wird mit ε bezeichnet. Im Folgenden betrachten wir nur noch nullterminierte Strings. Sei also S ein String der Länge n und $i, j \in [1, n]$. Wir bezeichnen mit

- $S[i]$ den i-ten Buchstaben von S.
- $|S|$ die Länge des Strings S.
- S^R den reversen String von S, also $S^R \coloneqq S[n]S[n-1]\cdots S[1]$.

- $S[i..j]$ den Substring von S der an i-ter Position startet und an j-ter Position endet.
  Wir definieren $S[i..j] = \varepsilon$ falls $i > j$, und definieren $S[i..j) \coloneqq S[i..j-1]$.
- S_i das Suffix von S das an der i-ten Position von S startet, d.h. $S_i = S[i..n]$.
- $S_i <_{\mathsf{lex}} S_j$ falls das Suffix S_i lexikographisch kleiner ist als S_j,
  d.h. es gibt ein $k \geq 0$ mit $S[i..i+k) = S[j..j+k)$ und $S[i+k] < S[j+k]$.

Neben den normalen Stringdefinitionen benötigen wir noch weitere Operationen auf Bitvektoren / Strings. Sei dazu S ein String / Bitvektor der Länge n, c ein Zeichen aus S und $i \in [0, n]$ ein Integer.

- Mit $\mathsf{rank}_S(c, i)$ bezeichnen wir die Anzahl der Vorkommen des Zeichens c im Präfix $S[1..i]$, d.h. $\mathsf{rank}_S(c, i) = \sum_{j=1}^{i} \mathbf{1}_{S[j]=c}$
- Mit $\mathsf{select}_S(c, i)$ bezeichnen wir die Position des i-ten Vorkommens von c in S, d.h. $\mathsf{select}_S(c, i) = \min\{\ j \in [0, n] \mid \mathsf{rank}_S(c, j) = i\ \}$.
- Mit $\mathsf{C}[c]$ bezeichnen wir die Anzahl der Zeichen in S die kleiner als c sind, d.h. $\mathsf{C}[c] = \sum_{i=1}^{n} \mathbf{1}_{S[i]<c}$

Wir bemerken, dass die Operationen rank und select auf Bitvektoren in $O(1)$ Zeit und linearem Speicheraufwand berechenbar sind [9], während die Operationen auf gewöhnlichen Strings mithilfe von *Wavelet Trees* in $O(\log |\Sigma|)$ Zeit und linearem Speicheraufwand berechenbar sind [6], wobei die dafür benötigten Datenstrukturen in einem Vorverarbeitungsschritt in linearer Zeit berechenbar sind. Das entsprechende C-Array kann mittels eines Scans des Strings zur Bestimmung der Buchstabenhäufigkeiten und anschließender kumulativer Summierung berechnet werden.

2.1 Wavelet Trees

Wavelet Trees sind eine sehr nützliche Datenstruktur, mithilfe derer rank und select effizient ermöglicht werden. Die Konstruktion eines Wavelet Trees für einen String S der Länge n mit Alphabetgröße σ benötigt $O(n \log \sigma)$ Zeit. Hierbei ist zu bemerken, dass der Wavelet Tree eine statische Datenstruktur ist, d.h. Änderungen der zugrundeliegenden Sequenz erfordern die Konstruktion eines neuen Wavelet Trees. Der Wavelet Tree bietet (unter anderem) folgende Operationen an:

- Elementzugriff $S[i]$ in $O(\log \sigma)$ Zeit
- Rankanfragen $\mathsf{rank}_S(c, i)$ in $O(\log \sigma)$ Zeit
- Selectanfragen $\mathsf{select}_S(c, i)$ in $O(\log \sigma)$ Zeit

- Die getIntervals - Methode. Der Aufruf von $\mathsf{getIntervals}(i, j)$ liefert hierbei die Menge

  $$\{ \langle c, lb, rb\rangle \mid c \in \{S[i], S[i{+}1], \ldots, S[j]\}, lb = \mathsf{rank}_S(c, i{-}1) \text{ und } rb = \mathsf{rank}_S(c, j)\}$$

  Anderst gesagt berechnet die getIntervals - Methode für jeden im Teilstring $S[i..j]$ auftretenden Buchstaben c:

  - Die Anzahl der Vorkommen von c im String $S[1..i)$
  - Die Anzahl der Vorkommen von c im String $S[1..j]$

  Die Methode benötigt dazu $O(z + z\log(\frac{\sigma}{z})) = O(z \log \sigma)$ Zeit, wobei z der Mächtigkeit der Menge entspricht [4, Lemma 3]. Die Methode wird in der `sdsl-lite`-Library [5] in Form einer `interval_symbols` - Methode bereitgestellt, die die Tupel nicht als Menge, sondern als nach Buchstabenwert aufwärts sortierte Tupelsequenz bereitstellt.

2.2 Suffixarray und Burrows-Wheeler-Transformation

Zunächst wollen wir einige wichtige Definitionen aus dem Bereich der Sequenzanalyse einführen.

i	$\mathsf{L}[i]$	sorted suffixes
1	A	$
2	C	$
3	A	$
4	C	A$
5	G	A$
6	$	AGCA$
7	$	AGGTGC$
8	G	C$
9	G	CA$
10	T	GA$
11	T	GC$
12	A	GCA$
13	$	GGTGA$
14	A	GGTGC$
15	G	GTGA$
16	G	GTGC$
17	G	TGA$
18	G	TGC$

Figure 1: Multi-String BWT für die Strings $S_1 =$ `AGCA$`, $S_2 =$ `AGGTGC$` und $S_3 =$ `GGTGA$`.

Definition 1. Sei S ein String der Länge n. Das *Suffix Array* [7] SA von S ist eine Permutation der Zahlen der Menge $[1, n]$ so, dass $S_{\mathsf{SA}[1]} <_{\mathsf{lex}} \cdots <_{\mathsf{lex}} S_{\mathsf{SA}[n]}$ gilt.

Definition 2. Sei S ein String der Länge n, und SA das zugehörige Suffixarray. Die Burrows-Wheeler-Transformation (BWT) [1] von S ist ein String L der Länge n der definiert ist als $\mathsf{L}[i] \coloneqq S[\mathsf{SA}[i] - 1]$ falls $\mathsf{SA}[i] > 1$ und $\mathsf{L}[i] \coloneqq \$$ falls $\mathsf{SA}[i] = 1$. Weiter ist die F-Spalte F definiert als ein String der Länge n mit $\mathsf{F}[i] \coloneqq S[\mathsf{SA}[i]]$.

Neben dem normalen Suffixarray wird auch gerne das Suffixarray von mehreren Strings $S_1, \ldots, S_m$ betrachtet. Dieses ergibt sich eigentlich analog zum normalen Suffixarray, indem wir die Strings zu einem neuen String $S = S_1\$S_2\$\cdots S_m\$$ verbinden, und dessen sortierte Suffixe beim Erreichen eines Trennsymbols quasi "abschneiden". Ein Beispiel eines sogenannten Multi-String Suffixarrays mit zugehöriger BWT ist in Abbildung 1 zu finden.

Die BWT erweist sich als sehr nützlich für Sequenzanalyse, da Sie mithilfe von rank- und select z.B. Operationen wie Patternsuche eines Patterns P in Zeit $O(|P| \cdot \log|\Sigma|)$ erlaubt, d.h. die Patternsuchzeit ist unabhängig von der Größe des zu untersuchenden Strings (mehr dazu gibt es in der Vorlesung "Algorithmen zur Sequenzanalyse"). Zentral für diese Eigenschaften ist der sogenannte Rückwärtsschritt.

Nehmen wir an, wir sind an einer Position i im Suffixarray und wollen die Position j berechnen, an der das vorangehende Suffix $(\mathsf{SA}[i]-1)$ zu finden ist. Sei $\mathsf{L}[i]$ das k-te Auftreten von $\mathsf{L}[i]$ in L. Dann entspricht die Position j gerade derjenigen Position, bei der in F das k-te Auftreten von $\mathsf{L}[i]$ zu finden ist.

Somit kann j wie folgt berechnet werden:

$$j \leftarrow \underbrace{\mathsf{C}[\mathsf{L}[i]]}_{\text{kleinere Buchstaben in F überspringen}} + \underbrace{\mathsf{rank}_{\mathsf{L}}(\mathsf{L}[i], i)}_{\text{zum } k\text{-ten Vorkommen in F voranschreiten}}$$

Indem wir also beim kürzesten Suffix anfangen, sukzessive die obere Regel anwenden und dazwischen immer den aktuellen Buchstaben $\mathsf{L}[i]$ ausgeben, erhalten wir den Originaltext in reverser Reihenfolge zurück (auch hierzu mehr in der Vorlesung "Algorithmen zur Sequenzanalyse").

2.3 FM-Index

Die Kombination von Wavelet Tree (Abschnitt 2.1) und BWT (Abschnitt 2.2) ergibt einen sehr kompakten und funktionsstarken Volltextindex, der gemäß seiner Erfinder Paolo Ferragina und Giovanni Manzini als FM-Index bezeichnet wird [3].

Der besagte FM-Index bietet hierbei neben der Wiederherstellung des Originaltextes auch viele weitere Operationen wie z.B. effiziente Patternsuche (auch approximativ), Ermittlung des k-mer Spektrums etc. an.

Der Grund für die sinnvolle Vereinigung ist darin begründet, dass ein Rückwärtssuchschritt in der BWT mithilfe der effizienten rank-Berechnungen des FM-Index sehr schnell durchgeführt werden kann: Hierzu ist neben des Wavelet Trees einer BWT lediglich das C - Array der BWT nötig.

Wir wollen uns nochmal kurz klarmachen, welche Rolle die getIntervals - Methode in diesem Kontext spielt. Sagen wir, wir haben die linke Grenze lb und die rechte Grenze rb eines ω-Intervals im Suffixarray bestimmt. Ein Aufruf von $M \leftarrow \mathsf{getIntervals}(lb, rb)$ sowie der Listenmodifikation

$$\langle c, lb, rb\rangle \rightarrow \langle c, \mathsf{C}[c] + lb + 1, \mathsf{C}[c] + rb\rangle$$

liefert uns dann die Menge aller $c\omega$ - Intervalle im Suffixarray.

Beispiel 3. Wir betrachten das Suffixarray aus Abbildung 1 Sei $\omega = \texttt{GC}$. Dessen Grenzen lauten $lb = 11$ und $rb = 12$. Ein Aufruf von $\mathsf{getIntervals}(lb, rb)$ liefert uns nun die Menge

$$M = \{\langle \texttt{A}, 2, 3\rangle, \langle \texttt{T}, 1, 2\rangle\}$$

Weiter gilt $\mathsf{C}[\texttt{A}] = 3$ sowie $\mathsf{C}[\texttt{T}] = 16$. Damit lautet M nach obiger Listenmodifikation wie folgt:

$$M = \{\langle \texttt{A}, 6, 6\rangle, \langle \texttt{T}, 18, 18\rangle\}$$

Diese Intervalle entsprechen nun gerade allen $c\omega$ - Intervallen, in diesem Fall also dem `AGC` - Intervall mit Grenzen 6 und 6 sowie dem `TGC` - Intervall mit den Grenzen 18 und 18.

3 Runminimierung

In einem von Bastien Cazaux und Eric Rivals 2018 erschienen Paper-Preprint wurde die Idee der Runminimierung in einer Multi-String BWT beschrieben [2]. Bevor wir jedoch die Idee dahinter beschreiben, soll zunächst der Begriff eines Runs in einer BWT definiert werden:

Definition 4 (BWT Run)**.** Sei L eine BWT mit Länge n. Dann bezeichnen wir einen nicht-leeren Substring $\mathsf{L}[i..j]$ von L als Run, falls der Substring die folgenden Eigenschaften erfüllt:

- $i = 1$ oder $S[i-1] \neq S[i]$
- $j = n$ oder $S[j+1] \neq S[j]$
- $S[i] = S[i+1] = \cdots = S[j]$

Anderst gesagt ist ein Run also ein längenmaximaler Substring in L der aus dem identischen Buchstaben besteht. Runs sind z.B. nützlich um eine BWT zu komprimieren: Indem jeder Run mittels Lauflängenkodierung kodiert wird (siehe Vorlesung "Datenkompression") kann die BWT meist sehr kompakt kodiert werden.

Die Idee der in [2] beschriebenen Idee lautet nun wie folgt: Wenn in einer Multi-String BWT mehrere identische sortierte Suffixe aufeinander folgen, so können die entsprechenden vorangestellten Buchstaben in der BWT beliebig vertauscht werden ohne die Rückwärtsschritteigenschaft der BWT zu verändern. Das heißt, das z.B. in der Multi-String BWT aus Abbildung 1 die Buchstaben $\mathsf{L}[4] = \texttt{C}$ und $\mathsf{L}[5] = \texttt{G}$ vertauscht werden können, da beide Buchstaben vom gleichen Suffix `A$` gefolgt werden. Diese Eigenschaft kann ausgenutzt werden, um Buchstaben mit gleichen Suffixen in der BWT derart zu permutieren, dass die Gesamtanzahl der Runs (und damit auch die Komprimierungsgröße einer lauflängenkodierten BWT) minimiert wird. Bevor wir jedoch den entsprechenden Ansatz beschreiben, soll zuerst geklärt werden, wie wir bestimmen können ob zwei Positionen i und j einer Multi-String BWT vom gleichen Suffix gefolgt werden.

3.1 Suffixtransitionsberechnung

In diesem Abschnitt befassen wir uns zuerst mit der Berechnung von Suffixtransitionen. Suffixtransitionen sind dabei Stellen, an denen das sortierte Suffix der Multi-String BWT quasi wechselt.

Die Idee hierzu stammt aus dem Kontext von BWT-basierten Triierepräsentationen [8, 10]: Man beginnt zunächst mit dem Suffixintervall \$ und markiert dort die oberste Position in einem seperaten Bitvektor B.[1] Nun wird das Suffixintervall durch einen Aufruf der Methode getIntervals und der in Abschnitt 2.3 beschriebenen Listenmodifikation zu Intervallen mit Suffixlänge 2 erweitert, im laufen Beispiel also zum A\$-Intervall $[4, 5]$ sowie dem C\$-Intervall $[8, 8]$, und dort auch weiter entsprechend der oberste Eintrag in B markiert. In dem dies sukzessive so fortgeführt wird, erhalten wir am Ende einen Bitvektor B, der uns genau die gewünschten Suffixtransitionen beschreibt.

Um aber nicht endlos Intervalle zu generieren bemerken wir, dass ein Intervall nur dann erweitert werden soll, wenn der Erweiterungsbuchstabe c kein Trennsymbol \$ ist: Ist der Buchstabe ein Trennsymbol, haben wir sozusagen das Ende des Strings erreicht, und können die Intervallgenerierung stoppen. Algorithmus 1 beschreibt das konkrete Verfahren um den Bitvektor B zu berechnen, ein Beispiel für den Bitvektor mit Suffixtransitionen ist in Abbildung 2 zu finden.

```
   Data: Multi-String BWT L in Form eines Wavelet Tree
   Result: Bitvektor B mit markierten Suffixtransitionen.
 1 initialize an empty bitvector B of size n + 1 filled with zeros
 2 B[n + 1] ← 1
 3 initialize an empty queue Q
 4 push interval [1, rank_L(\$, n)] onto Q
 5 while Q is not empty do
 6 |   pop first element [lb, rb] from queue Q
 7 |   B[lb] ← 1
 8 |   M ← getIntervals(lb, rb)
 9 |   foreach ⟨c, l, r⟩ ∈ M with c ≠ \$ do
10 |     push element [C[c] + l + 1, C[c] + r] onto Q
11 return B
```

Algorithm 1: Berechnung der Suffixtransitionen einer Multi-String BWT.

Die Laufzeit von Algorithmus 1 ergibt sich aus der Zahl der generierten Intervalle mal dem Aufwand ein Intervall zu generieren, im Fall eines Wavelet Trees also zu $O(n \log \sigma)$.

3.2 High-Level Runminimierung

Nachdem im letzten Abschnitt ein Ansatz zur Bestimmung von Suffixtransitionen vorgestellt wurde wollen wir uns nun damit beschäftigen, wie Buchstaben von ω-Intervallen so permutiert werden können, dass die Anzahl der Runs einer Multi-String BWT minimiert werden können.

[1] Im Beispiel aus Abbildung 1 wäre das erste Intervall also $[1, 3]$, und der Eintrag $B[1]$ würde auf 1 gesetzt

i	B	$\mathsf{L}[i]$	sorted suffixes
1	1	A	$
2	0	C	$
3	0	A	$
4	1	C	A$
5	0	G	A$
6	1	$	AGCA$
7	1	$	AGGTGC$
8	1	G	C$
9	1	G	CA$
10	1	T	GA$
11	1	T	GC$
12	1	A	GCA$
13	1	$	GGTGA$
14	1	A	GGTGC$
15	1	G	GTGA$
16	1	G	GTGC$
17	1	G	TGA$
18	1	G	TGC$

Figure 2: Multi-String BWT für die Strings $S_1 =$ AGCA\$, $S_2 =$ AGGTGC\$ und $S_3 =$ GGTGA\$, sowie Markierung der Suffixtransitionen im Bitvektor B.

Da die Problemlösung nicht ganz trivial ist, wollen wir uns das Problem zunächst ganz abstrakt vorstellen: Wir stellen uns die ω-Intervalle als Knoten in einem gerichteten Pfad dar. Die Kanten zwischen den Knoten sind dabei dann quasi die Übergänge von einem ω-Intervall zum nächsten. Entsprechend der BWT enthält jeder Knoten eine Menge von Buchstaben, im Konkreten also diejenigen Buchstaben die im entsprechenden Intervall in der BWT stehen.

Nun kann für jede Kante der Schnitt I zwischen beiden Knoten berechnet werden. Die Aufgabe kann dann wie folgt beschrieben werden: Bestimme für jede Kante eine Beschriftung aus der Menge des entsprechenden Schnittes I, sodass zwei aufeinanderfolgende Kanten nur dann dieselbe Beschriftung haben, falls der dazwischenliegende Knoten nur genau einen Buchstaben enthält. Da dies abhängig von der Eingabe nicht immer gewährleistet werden kann, können Kanten auch mit $\perp$ beschriftet werden.

Eine Minimierung der Runs entspricht dann einer Minimierung der Kantenbeschriftung $\perp$, da für jede Kantenbeschriftung $c \neq \$$ die Übergänge zwischen den Suffixintervallen mit der Beschriftung der Kante vorgenommen werden können. Ein Beispiel der eben geschilderten Abstraktion ist in Abbildung 3 zu finden, wir wollen nun im weiteren einen Algorithmus entwickeln, der uns die geforderte Kantenbeschriftung erzeugt.

Die Idee des Algorithmus wird es sein, jeweils die Schnittmengen auf den Kanten zu vergleichen und dadurch eine Fallunterscheidung durchzuführen. Dafür benötigen wir zunächst folgende Funktionen:

- $\mathsf{next}(v)$: Gibt den auf v folgenden Knoten im Pfad an
- $\mathsf{chars}(v)$: Gibt die Zeichen des Knotens v als Menge aus

Eine Implementierung der angegebenen Funktionen verschieben wir auf später, zuvor nun zur eigentlichen Idee: Nehmen wir an, wir sind bei einem Knoten v mit Folgeknoten $v_{\mathsf{next}} \coloneqq \mathsf{next}(v)$ und Schnittmenge $I \coloneqq \mathsf{chars}(v) \cap \mathsf{chars}(v_{\mathsf{next}})$. Sei

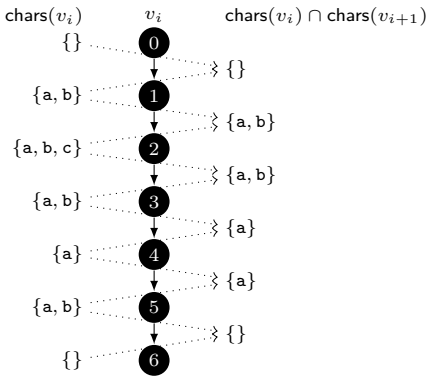

Figure 3: Abstrahierung der Kantenminimierung: Jeder Knoten des Pfades entspricht einem ω-Intervall, die Kanten symbolisieren Übergänge zwischen den Intervallen. Kanten werden mit Buchstaben aus der Schnittmenge beider Knoten beschriftet, wobei zwei aufeinanderfolgende Kanten nur dann gleich beschriftet werden dürfen, wenn der dazwischenliegende Knoten nur ein Symbol besitzt. Alle Kanten die aufgrund obriger Regel nicht beschriftet werden können, werden mit einem $\perp$ versehen.

weiter die Folgeschnittmenge definiert als $I_{\mathsf{next}} \coloneqq \mathsf{chars}(v_{\mathsf{next}}) \cap \mathsf{chars}(\mathsf{next}(v_{\mathsf{next}}))$. Wir unterscheiden die folgenden Fälle:

- $|I \cap I_{\mathsf{next}}| = 0$
  Dann beeinflussen wir durch die Wahl der Kantenbeschriftung der Kante (v, v_{next}) den weiteren Verlauf nicht, und können so eine beliebige Beschriftung $c \in I$ für die Kante wählen.

- $|I \cap I_{\mathsf{next}}| = 1$
  Hier beeinflusst die Wahl der Kantenbeschriftung für die Kante (v, v_{next}) möglicherweise den weiteren Verlauf. Im Falle $|I| = 1$ haben wir keine Wahl, und müssen die Kante mit dem Buchstaben $c \in I$ beschriften, da wir sonst einen Run mehr als nötig generieren würden. Im Fall $|I| > 1$ können wir die Kante dagegen mit einem beliebigen Buchstaben aus der Menge $I \setminus I_{\mathsf{next}}$ beschriften ohne den weiteren Verlauf zu beeinflussen.

- $|I \cap I_{\mathsf{next}}| > 1$
  Dies ist der komplizierteste Fall, da wir durch eine ungeschickte Kantenwahl hier die Zahl der Runs erhöhen könnten. Wir gehen hier wie folgt vor: Wir bestimmen zwei Buchstaben $c_0, c_1 \in I \cap I_{\mathsf{next}}$ und folgen dem Pfad solange weiter, bis ein Knoten u erreicht wird, in dessen Schnittmenge I_u mit dessen Folgeknotens $\mathsf{next}(u)$ nicht mehr beide Buchstaben enthalten sind, d.h. $\{c_0, c_1\} \not\subseteq I_u$. Anschließend können die Kanten auf dem Pfad von v nach u alternierend mit den Buchstaben c_0 und c_1 beschriftet werden. Durch eine geschickte Wahl des Anfangsbuchstabens (c_0 oder c_1)

kann so sichergestellt werden, dass jede Kante beschriftet, und der weitere Verlauf nicht beeinflusst wird.

Statt nun weiter auf alle Spezialfälle näher einzugehen, wollen wir lieber die Idee in Form von Pseudocode in Algorithmus 2 wiedergeben:

```
   Data: Pfad mit Knoten (ω-Intervallen) und Kanten (Übergängen zwischen den
         ω-Intervallen).
   Result: Kantenbeschriftung zwischen den Knoten.

 1 v ← first node in path
 2 I ← chars(v) ∩ chars(next(v))
 3 while v is not the last node in path do
 4     v_next ← next(v)
 5     I_next ← chars(v_next) ∩ chars(next(v_next))
 6     if |I ∩ I_next| = 0 then
 7         if |I| = 0 then
 8             label outgoing edge of v with ⊥
 9         else
10             label outgoing edge of v with any c ∈ I
11     else if |I ∩ I_next| = 1 then
12         if |I| = 1 then
13             label outgoing edge of v with any c ∈ I
14             if |chars(v_next)| > 1 then
15                 I_next ← I_next \ I
16         else
17             label outgoing edge of v with any c ∈ I \ I_next
18     else
19         let c_0, c_1 ∈ I ∩ I_next be two different elements
20         i ← 1
21         do
22             v_next ← next(v_next)
23             I_next ← chars(v_next) ∩ chars(next(v_next))
24             i ← i + 1
25         while {c_0, c_1} ⊆ I_next
26         if |chars(v_next)| = 1 then
27             if (chars(v_next) = {c_0} and i is odd)
                   or (chars(v_next) = {c_1} and i is even) then
28                 swap variables c_0 and c_1
29         else if c_{i mod 2} ∈ I_next then
30             swap variables c_0 and c_1
31         for j ← 1 to i do
32             label outgoing edge of node v with character c_{j mod 2}
33             v ← next(v)
34     v ← v_next
35     I ← I_next
```

Algorithm 2: Kantenbeschriftung mit minimalen ⊥-Zeichen berechnen.

Der erste beschriebene Fall ist soweit eigentlich selbsterklärend: Sind die Schnittmengen disjunkt (Zeile 6), so kann ein beliebiges Element aus I für die Kantenbeschriftung genutzt werden (sofern I nicht-leer ist).

Im zweiten Fall (Schnittmengen teilen sich ein gemeinsames Element, Zeile 11) muss im Fall $|I| = 1$ der entsprechende Buchstabe zur Kantenbeschriftung genutzt werden. Weiter muss dann überprüft werden, ob der Folgeknoten mehr als einen Buchstaben besitzt (Zeile 14): In diesem Fall muss dann nämlich der

Buchstabe aus der nächsten Schnittmenge entnommen werden, da sonst das ω-Intervall mit dem gleichen Buchstaben beginnen und enden würde, und wir so die Zahl der Runs erhöhen würden. Im Fall $|I| > 1$ hingegen können wir ein Element $c \in I \setminus I_{\mathsf{next}}$ benutzen, um so die weiteren Kantenschriftungen nicht zu beeinflussen.

Zuletzt wollen wir noch den dritten Fall anschauen, also dass die beiden Schnittmengen mehr als ein Element gemeinsam haben (Zeile 18). Hier bestimmen wir dann wie besprochen zwei Elemente c_0 und c_1 des Schnittes (Zeile 19), gehen solange im Pfad weiter bis eine Schnittmenge diese Elemente nicht mehr enthält (Zeilen 21-25) und beschriften den ganzen betrachteten Teilpfad alternierend mit c_0 und c_1 (Zeilen 31-33). Um die alternierende Beschriftung richtig "auszurichten" prüfen wir zum einen, ob der Endknoten des Teilpfades aus nur einem Buchstaben besteht (Zeile 26). Da die alternierende Beschriftung immer mit c_1 beginnt, bedeutet dies, dass ein Teilpfad mit gerader Länge immer mit c_0 endet, ein Teilpfad mit ungerader Länge hingegen mit c_1. Besteht der Endknoten nur aus einem Buchstaben, der dann auch noch entweder c_0 oder c_1 ist, so wird durch die Zeilen 27-28 sichergestellt, dass die Beschriftung auch gerade mit diesem Buchstaben aufhört. Ist es hingegen der Fall, dass die Endbeschriftung gerade in der darauffolgenden Schnittmenge enthalten ist (Zeile 26), so ändern wir die Alternierung, um die Schnittmenge I_{next} nicht zu verkleinern (Zeile 27).

3.3 Implementierung der Hilfsfunktionen

Nachdem wir im letzten Abschnitt einen High-Level Algorithmus zur Runminimierung kennen gelernt haben ist nun die Frage, wie die einzelnen Operationen des Algorithmus effizient implementiert und wie daraus eine neue Multi-String BWT erzeugt werden können.

Fangen wir dazu zunächst mit den Knoten in Algorithmus 2 an. Wie bereits erwähnt symbolisiert ein Knoten ein Suffixintervall, weswegen wir Knoten am besten durch Intervalle $[i, j]$ in der Multi-String BWT darstellen können: zum Beispiel kann der Knoten für das `A$`-Intervall aus Abbildung 1 durch das Intervall $[4, 5]$ dargestellt werden.

next - Funktion

Mithilfe dieser Knotenrepräsentation ist auch unmittelbar klar, wie wir die Operation next implementieren können: Hierzu nutzen wir den in Abschnitt 3.1 vorgestellten Bitvektoren B, laufen von der übergebenen rechten Grenze solange weiter, bis wir das Ende des entsprechenden Intervalls finden, siehe hierzu Algorithmus 3.

```
Data: Bitvektor B mit Suffixtransitionen aus Abschnitt 3.1, Knotenintervall [i, j] eines
      Knotens v.
Result: Knotenintervall [i_next, j_next] des Folgeknotens next(v).

1 function next([i, j])
2     i_next ← j + 1
3     j_next ← j + 1
4     while j_next ≤ n and B[j_next + 1] = 0 do
5         j_next ← j_next + 1
6     return [i_next, j_next]
```

Algorithm 3: Implementierung der next-Funktion aus Algorithmus 2.

Dadurch das wir im Abschnitt 3.1 den Eintrag $B[n+1]$ auf 1 gesetzt haben, ist auch sichergestellt, das wir als letzten Knoten im Pfad einen einen leeren Knoten erreichen und so die Abbruchbedingungen für Algorithmus 2 korrekt eingehalten werden.

chars-Funktion und Mengenrepräsentation

Nachdem wir nun also über die Knoten des Pfades iterieren können, stellt sich als nächstes die Frage, wie man an Mengen von Zeichen eines Knotens ermitteln und auch Schnittmengen bilden kann. Wir kümmern uns zuerst darum, für einen Knoten die Menge seiner beinhalteten Zeichen zu ermitteln. Hierbei hilft uns die getIntervals-Methode aus Abschnitt 2.1, da Sie die Buchstaben unter anderem die Buchstaben eines Intervalls in aufsteigender Reihenfolge liefert. Um Buchstabenmengen darzustellen benutzen wir dementsprechend einfache Strings, in denen wir die entsprechenden Buchstaben in aufsteigender lexikographischer Reihenfolge abspeichern.

```
Data: Multi-String BWT L in Form eines Wavelet Trees, Knotenintervall [i, j] eines
      Knotens v.
Result: Knotenintervall [i_next, j_next] des Folgeknotens next(v).

1 function chars([i, j])
2     M ← getIntervals(i, j)
3     foreach ⟨c, rb, lb⟩ ∈ M in ascending character order do
4         append character c to a string S
5     return ⟨S, |M|⟩
```

Algorithm 4: Implementierung der chars-Funktion aus Algorithmus 2.

Algorithmus 4 zeigt, wie der entsprechende String (als Paar mit der Anzahl der Buchstaben) erzeugt werden kann. Tatsächlich muss aber im Falle der `sdsl-lite` - Library gar keien Konvertierung stattfinden, da die Bibliothek mit der `interval_symbols` - Funktion die getIntervals-Funktion bereitstellt: die Ausgabe besteht aus drei Arrays (Zeichen, linke Grenze, rechte Grenze), dementsprechend muss nur das erste Array sowie die Größe der Ausgabe benutzt werden.

Mengenschnitte und Differenzen

In Algorithmus 2 werden oft Mengenschnitte, Mengendifferenzen und Teilmengenoperationen benutzt. Da wir in unserem Fall Mengen als Strings mit aufsteigend

geordneten Buchstabenwerten darstellen, ist die Frage, wie diese Operationen effizient gestaltet werden können.

Hierzu benutzten wir eine Idee, die der Idee des Merge-Schrittes im Mergesort-Algorithmus sehr nahe kommt: da zwei Mengen immer aus aufsteigend sortierten Buchstaben bestehen, können wir mittels zwei Zeigern in beiden Strings immer Buchstaben gegeneinander abgleichen, und so in linearer zeit sowohl Schnittmenge als auch beide Mengendifferenzen berechnen. Die konkrete Vorgehensweise wird in Algorithmus 5 aufgezeigt.

```
   Data: Zwei Buchstabenmengen A und B in der Form A≙⟨S, n⟩ und B≙⟨T, m⟩.
   Result: Buchstabenmengen A ∩ B, A \ B und B \ A.
 1 function intersection(⟨S, n⟩, ⟨T, m⟩)
 2     initialize an empty string I of size min{n, m}
 3     i ← 1                              // input position in set A
 4     j ← 1                              // input position in set B
 5     p ← 0                  // output position in set A \ B stored in S
 6     q ← 0                  // output position in set B \ A stored in T
 7     r ← 0                  // output position in set A ∩ B stored in I
 8     while i ≤ n and j ≤ m do
 9         while i ≤ n and (j > m or S[i] < T[j]) do
10             p ← p + 1
11             S[p] ← S[i]
12             i ← i + 1
13         while j ≤ m and (i > n or S[i] > T[j]) do
14             q ← q + 1
15             T[q] ← T[j]
16             j ← j + 1
17         if i ≤ n and j ≤ m and S[i] = T[j] then
18             r ← r + 1
19             I[r] ← S[i]
20             i ← i + 1
21             j ← j + 1
22     return ⟨I, r⟩, ⟨S, p⟩, ⟨T, q⟩                    // A ∩ B, A \ B, B \ A
```

Algorithm 5: Schnittmenge und Mengendifferenzen zweier Buchstabenmengen berechnen.

Mit der Methode aus Algorithmus 5 lassen sich so alle gewünschten Mengenoperationen aus Algorithmus 2 umsetzen. Zum Beispiel kann die Teilmengenoperation $\subseteq$ implementiert werden, indem man die Methode mit den beiden Mengen aufruft, und den entsprechenden Schnitt (oder die Mengendifferenzen) überprüft. Weiter empfiehlt es sich bei der genannten Mengenrepräsentation die "zufälligen" Elemente in Algorithmus 2 immer als die vordersten ein bzw zwei Zeichen zu nehmen, da so die Strings nicht modifiziert werden müssen, und nur die Größenanzahl auf eins bzw. zwei reduziert werden kann.

Zur Überprüfung, ob ein Element in der Menge enthalten ist (Zeile 29) empfiehlt sich entweder eine lineare Suche, oder aber auch eine binäre Suche. Zuletzt wollen wir darauf hinweisen, dass bei der Implementierung der Methode nicht jedesmal neue Strings erzeugt werden sollten, da dies unter Umständen aufwendig ist. Stattdessen sollte die Methode so implementiert werden, dass 5 Strings der Mindestgröße σ (in unserem Fall also 256) sowie 5 Integervariablen per Referenz übergeben werden. Während dann die ersten zwei Paare den Eingabemengen entsprechen, können die Ausgabemengen in den referen-

zierten Strings platziert und deren Integer entsprechend modifiziert werden.

Kantenbeschriftung verarbeiten und Multi-String BWT erzeugen

Zuletzt wollen wir uns noch damit auseinandersetzen, was die Kantenbeschriftung bedeutet, und wie dies uns hilft die gesuchte Multi-String BWT zu erzeugen. Wir bemerken, dass die Beschriftung einer ausgehenden Kante eines Knotens gerade bedeutet, dass im entsprechenden Intervall des Knotens in der BWT der zur Beschriftung genutzte Buchstabe als letztes geschrieben werden muss.

Da in Algorithmus 2 die Kanten nun gerade von oben nach unten nacheinander beschriftet werden, können wir in einer globalen Variablen die Buchstabenbeschriftung der letzten geschriebenen Kante speichern. Wir dann die nächste Kante beschriftet, wissen wir, dass der dazwischenliegende Knoten am Anfang das Zeichen der globalen Variablen und am Ende das soeben enthaltene Zeichen schreiben muss. Nachdem wir dann also die Einträge in der BWT gemäß dieser Bedingung herausgeschrieben haben, können wir uns wieder das Zeichen der Kante speichern, und so im nächsten Schritt wieder entsprechend fortfahren.

Indem wir so mit dem initiallen Buchstaben $\perp$ anfangen, den ersten Knoten in Algorithmus 2 als das Intervall $[1, \mathsf{rank}_{\mathsf{L}}(\$, n)]$ wählen, und den Algorithmus abbrechen, sobald wir beim letzten virtuellen Knoten angelangen (das ist gerade der $B[n+1]$-Eintrag im Bitvektor B), wird dabei die gesamte runminimierte BWT erzeugt. Algorithmus 6 zeigt das konkrete gewünschte Verhalten, das auch mit einer mit $\perp$ beschrifteten Kante kein Problem hat.

```
   Data: Multi-String BWT L in Form eines Wavelet Trees, globale Variablen L̂ (neue
         runminimierte BWT, intial leer) und c_last (Kantenbeschriftung der letzten Kante,
         initial ⊥) sowie Parameter [i, j] (aktueller Knoten) und c (Beschriftung der Kante).
   Result: Runminimierte BWT L̂ (nach Terminierung von Algorithmus 2).

 1 function labeloutedge([i, j], c)
 2     M ← getIntervals(i, j)
 3     split M into 3 arrays M_c (characters), M_lb (left bounds), M_rb (right bounds)
 4     i ← 1
 5     while i ≤ |M| do
 6         if M_c[i] = c and i ≠ |M| then
 7             swap variables M_c[i] and M_c[|M|]
 8             swap variables M_lb[i] and M_lb[|M|]
 9             swap variables M_rb[i] and M_rb[|M|]
10         else
11             if M_c[i] = c_last then
12                 swap variables M_c[i] and M_c[1]
13                 swap variables M_lb[i] and M_lb[1]
14                 swap variables M_rb[i] and M_rb[1]
15             i ← i + 1
16     for i ← 1 to |M| do
17         append (M_rb[i] − M_lb[i]) times the character M_c[i] to string L̂
18     c_last ← c
```

Algorithm 6: Runminimierte BWT durch Kantenbeschriftung erzeugen.

Zum Verständnis der Zeilen 5-15 von Algorithmus 6 noch folgendes: Nehmen wir an, wir scannen die Buchstaben und sehen, dass der Buchstabe, der ans Ende gehört, aktuell vorliegt (Zeile 6). Dann können wir den Buchstaben mit dem

letzten Buchstaben tauschen, da wir den letzten Buchstaben aber noch nicht untersucht haben, wird i nicht erhöht.

Liegt hingegen der Fall vor, dass wir den Buchstaben sehen, der an den Anfang des Arrays gehört, so tauschen wir dies auch entsprechend. Da wir diesen Buchstaben aber vorher schon geprüft haben, kann i um eins erhöht werden. Liegt keiner der beiden Fälle vor, kann i ebenfalls um eins erhöht werden.

Weiter wird die Anweisung in Zeile 3 bei der Umsetzung mithilfe der `sdsl-lite` Library nicht benötigt, da hier `interval_symbols` die Ausgabe entsprechend in 3 Arrays vornimmt.

4 Zusammenfassung und Ziele der Arbeit

- Implementierung der in diesem Konzept beschriebenen Algorithmen mithilfe der `sdsl-lite` Library
- Testen der Wirksamkeit der Methode anhand von bereitgestellten Testdaten
  - Anteilige Messung der Zahl von minimierten Runs
  - Vergleich einer lauflängenkodierten BWT mit und ohne Runminimierung: Bei der Lauflängenkodierung wird ein Run der Länge l in eine Sequenz der Länge $\lceil \log_2(l+1) \rceil$ umgewandelt (siehe Vorlesung "Datenkompression").
- Schreiben einer Ausarbeitung, in der die Ideen dieses Konzepts **in eigenen Worten** wiedergegeben werden sowie die Wirksamkeit der Methode präsentiert wird (ca. 10 - 20 Seiten)
- Präsentieren der Idee und der Ergebnisse in einer Abschlusspräsentation

References

[1] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[2] Bastien Cazaux and Eric Rivals. Strong link between BWT and XBW via Aho-Corasick automaton and applications to Run-Length Encoding. *CoRR*, abs/1805.10070, 2018.

[3] Paolo Ferragina and Giovanni Manzini. Indexing Compressed Text. *Journal of the ACM*, 52(4):552–581, 2005.

[4] Travis Gagie, Gonzalo Navarro, and Simon J.P̃uglisi. New Algorithms on Wavelet Trees and Applications to Information Retrieval. *Theoretical Computer Science*, 426, 2010.

[5] Simon Gog. `sdsl-lite` Library. `https://github.com/simongog/sdsl-lite`. last visited November 2018.

[6] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order Entropy-compressed Text Indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 841–850, 2003.

[7] Udi Manber and Gene Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 5:935–948, 1993.

[8] Giovanni Manzini. XBWT Tricks. In *Proceedings of the 23rd International Symposium on String Processing and Information Retrieval*, SPIRE '16, pages 80–92, 2016.

[9] J.Ian Munro. Tables. In *Foundations of Software Technology and Theoretical Computer Science*, FSTTCS '96, pages 37–42, 1996.

[10] Enno Ohlebusch, Stefan Stauß, and Uwe Baier. Trickier XBWT Tricks. In *String Processing and Information Retrieval*, SPIRE '18, pages 325–333, 2018.