Komprese dat pomocí transformace Burrows–Wheeler

Petr Dvořáček – xdvora0n@stud.fit.vutbr.cz

6. května 2014

Komprese pomocí Burrows–Wheelerovy transformace se řadí mezi blokové komprese. Tento algoritmus nachází využití i v bioinformatice pro sestavování genomu organismů z krátkých fragmentů (řetězců) DNA.

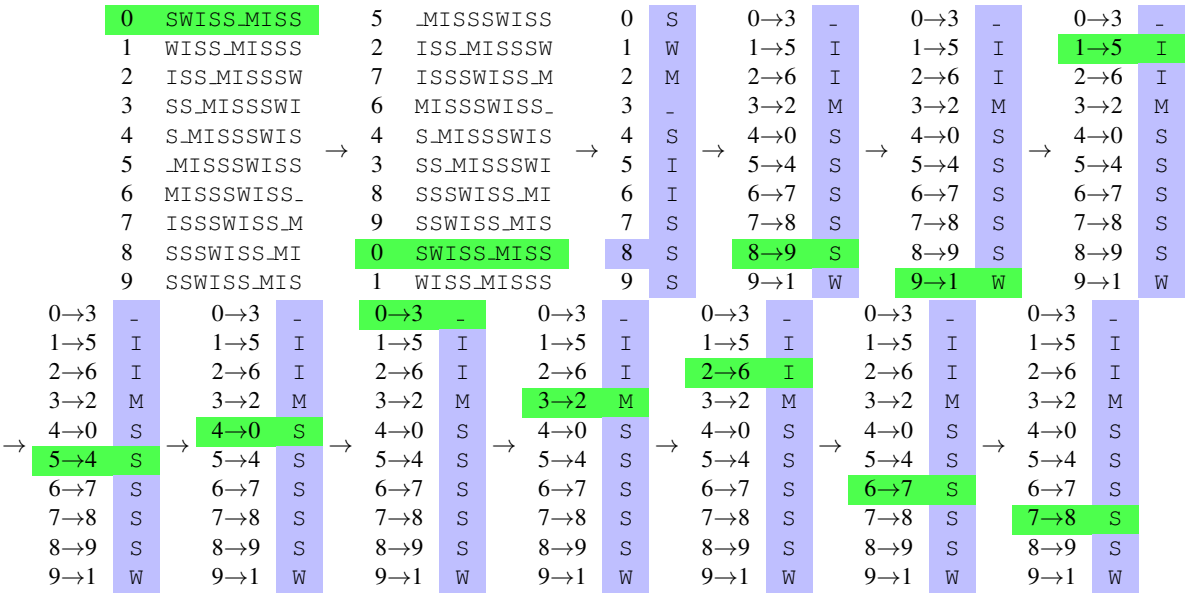
Tato práce se však zabývá prvním problémem a tou je komprese dat. Samotná komprese je rozdělena do několika kroků. Nejdříve se vstupní text ztransformuje pomocí Burrows–Wheelerovy transformace (dále jen BWT), jak je uvedeno v sekci 1. Výstup BWT zpracujeme pomocí funkce Move to Front (dále jen MFT). O tomto algoritmu pojednává sekce 2. Move to Front zvyšuje efektivnost entropického kódování. Data získána MTF jsou pak vstupem do algoritmu Run Length Encoding, který provádí zkrácení posloupnosti stejných bytů. Vizte sekci 3. Výstup RLE je nakonec zpracován pomocí entropického kódování. V tomto projektu jsem použil statické Huffmanovo kódovaní. O tom pojednávám v sekci 4. Následujou experimenty a závěr.

1 Burrows–Wheelerova transformace

Burrows wheelerova transformace vyrábí permutaci znaků z řetězce x o velikosti n. Snaží se znaky seřadit tak, abychom získali velkou posloupnost stejných znaků za minimální cenu (z pohledu paměti).

Nejdříve řetězec x, kde například x =SWISS_MISS, se uspořádá do matice $M = n \times n$. Na jednotlivých řádcích aplikujeme operátor rotace původnícho řetězce. Tyto rotované řádky se pak seřadí lexiograficky. Výstupem je pak poslední sloupec s indexem řádku i na původní řetězec x. Na příkladu je to znázorněno divně modrou barvou, kde $f(x)$ =SWM_SIISSS a $i = 8$.

Původní text získáme následujícím způsobem. Jednotlivá písmena se seřadí stabilním řadícím algoritmem. První písmeno původního textu se nachází v seřazeném poli na uložené pozici i. Nový index je pak hodnota indexu nalezeného písmena v neseřazeném poli. Ten pak určuje pozici druhého písmena v seřazeném poli. Analogicky získáme původní řetězec, tento postup je znázorněn níže:



2 Move to Front

Myšlenkou této části je, že se vytvoří seznam naposledy použitých symbolů. Vygeneruje se takovým způsobem, že bude obsahovat 256 položek, s hodnotami $(0, 1, 2, 3, \ldots, 254, 255)$. Seznam se aktualizuje tak, že se na jeho počátek dá element se vstupní hodnotou symbolu. Výstupem je pak index, kde byla hodnota nalezena. Inverzní operace je podbná s tím rozdílem, že vstupem je index do slovníku a výstupem je hodnota symbolu.

Příklad: Nechť máme seznam o 26 položkách s hodnotami $(A \ldots Z)$ a vstupní řetězec $x =$`SWMSIISSS`.

0. iterace	$(A, B, C, D, \ldots, Z)$	S	W	M	S	I	I	S	S	S
1. iterace	$(S, A, B, C, \ldots, Z)$	18	W	M	S	I	I	S	S	S
2. iterace	$(W, S, A, B, \ldots, Z)$	18	22	M	S	I	I	S	S	S
3. iterace	$(M, W, S, A, \ldots, Z)$	18	22	12	S	I	I	S	S	S
4. iterace	$(S, M, W, A, \ldots, Z)$	18	22	12	2	I	I	S	S	S
...	...					...				
9. iterace	$(S, I, M, W, \ldots, Z)$	18	22	12	2	8	0	1	0	0

3 Run Length Encoding

Tato část algoritmu se snaží komprimovat posloupnosti shodných znaků. Tato posloupnost je zakódovana do formátu *délka,znak*. Můžeme dosáhnout lepší komprese dat, zmenšujeme-li posloupnosti délky větší než dva. Pak použijeme formát *speciální znak, délka, znak* s důmyslným ošetřením výskytu speciálního znaku, např. jeho zdvojení.

Mějme například řetězec $x =$`#BAAANAAAANAAAAAAAAA`, kde speciální znak je `#`. Pak při použití RLE získáme posloupnost: $rle(x) =$`##B#3AN#4AN#9A`.

4 Statické Huffmanovo kódování

V tomto projektu bylo použito statické Huffmanovo kódovaní. Huffmanův strom byl sestrojen podle tabulky uvedné níže. Předpokládám, že nejvíce zastoupené symboly nesou malé hodnoty.

Listů	Hloubka	Prvky
2	2	0, 1
4	4	2, 3, 4 + 10 ta měla vyšší pravděpodobnost než 5 a 4.
8	6	5, 6, 7, 8, 9, 11, 12 + speciální znak 255
16	8	13, 14 atd. + maximální délka RLE 254
32	10	28, 29 ..
64	12	60, 61 ..
127	13	124, 125 ..
4	15	Prvky 251, 252, 253 + speciální znak konce bloku.

5 Experimentální vyhodnocení a závěr

Projekt byl naimplementován v jazyce C a byl otestován na školním serveru Merlin (škoda, že se nejmenuje Gandalf). Velikost bloku byla nastavena na 200000. Pro změnu stačí přeložit příkazem: `make BLOCK_SIZE=42`. Podařilo se komprimovat 751kB referenční soubor na 301kB.

Další prací do budoucna bych viděl v optimalizaci programu. Některé části (např. MTF) by se dali provést efektivnějším způsobem (např. pomocí `memcpy()`). Rovněž není možnost využití megabytových bloků, neboť by byla pro ně nutná alokace pomocí funkce `malloc()`.