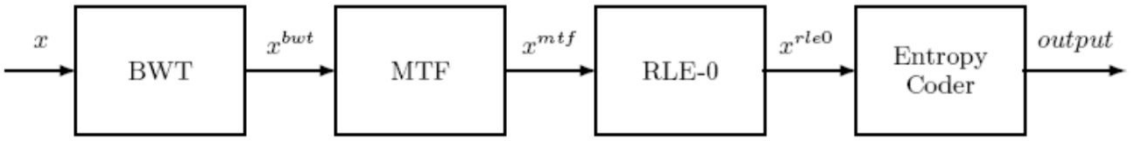
Zadání

V jazyce C/C++ implementujte knihovnu a aplikaci pro kompresi a dekompresi textových souborů s využitím BWT transformace. Aplikace bude mít následující strukturu:



Jednotlivé kroky představují: BWT transformaci, překódování řetězce znaků operací Move-to-front, efektivní reprezentaci delších sledů čísel metodou RLE a entropické kódování (statický, adaptivní Huffmanův kód nebo aritmetický kodér)

Řešení

Implementace řešení jednotlivých transformačních a kódovacích bloků je provedena v samostatných knihovnách. Ty lze nalézt ve složce `blocks`. Podrobnější popis funkce a implementace jednotlivých bloků je rozepsán dále. Společné typy lze dále nalézt v `blocks/common.hpp`. Knihovna pro `bwted` se nachází samostatně v kořenovém adresáři projektu.

Samotný soubor je pro účely kódování rozdělen a čten po blocích o velikosti `BLOCK_SIZE`. Tato velikost byla experimentálně stanovena na velikost 10000 znaků. Příliš velká velikost způsobuje zpomalení při Burrows-Wheelerově transformaci, jelikož je nutné tvořit dlouhé řetězce permutací. Naopak nedostatečná velikost navozuje příliš časté Huffmanovo kódování a tedy časté prokládání kódované zprávy hlavičkou s tabulkou frekvencí jednotlivých znaků. Program kóduje zprávy následujícím způsobem:

```
While (načti blok vstupního souboru o velikost BLOCK_SIZE popř. zbytek souboru)
       Transformuj pomocí Burrows-Wheelerovy transformace
       Transformuj Move-to-front
       Překóduj pomocí Run-length
       Zakóduj jako Huffmanovo kódování
       Zapiš výslednou zprávu do výstupu
       Aktualizuj logovací informace
```

Dekódování zprávy je provedeno analogicky, ovšem s odlišným způsobem čtení zakódovaného souboru. Ten má totiž jinou délku a není možné jej číst po blocích o velikosti `BLOCK_SIZE`, naopak je třeba jej zpracovávat po znacích, dokud není identifikován oddělovač `DELIMITER`. Postup dekódování je tedy:

```
While (načti znak, který určuje index pro Burrows-Wheelerovu t. a dokud není EOF)
       Načti znaky po nejbližší 2 oddělovače DELIMITER jako tabulku frekvencí
       Načti znaky po nejbližší 3 oddělovače DELIMITER jako kódovanou zprávu
       Aktualizuje logovací informace
       Dekóduj Huffmanovo kódování
       Dekóduj Run-length
       Transformuj zpět Move-to-front
       Transformuj zpět Burrows-Wheelerovu transformaci
       Zapiš výslednou dekódovanou zprávu
       Aktualizuj logovací informace
```

Burrows-Wheelerova transformace

Implementováno jako knihovna bwt. Tato transformace spočívá v zakódování řetězce jako permutace. Tato výsledná permutace se vytvoří ze všech permutací doleva rotovaného vstupního řetězce, které jsou seřazeny abecedně a následně je vybrán jejich poslední znak. Mimo této permutace je třeba si pamatovat i index, na kterém se nachází originální nerotovaný řetězec. Postupujeme tedy následovně:

```
Pro každou pozici v řetězci
      Zapamatuj si pozici (index) a vytvoř a zapamatuj si permutaci
Seřaď tyto permutace abecedně
Pro každou pozici N v seznamu permutací
      Na N-tou pozici ve výstupním řetězci ulož písmeno ze vstupního řetězce
            (na indexu permutace)
      Pokud je index permutace 0
            Ulož pozici N jako index pro nalezení originální permutace
```

Move-to-front transformace

Účelem této transformace je snížit hodnotu jednotlivých znaků tak. Místo uchovávání znaků zprávy tato transformace uchovává hodnoty indexů do abecedy a to tím způsobem, že pro každý znak si spočte, na jakém indexu v abecedě se nachází a následně jen v tomto poli abecedně seřazených symbolů zařadí na začátek. Vzniká tedy posloupnost indexů do abecedy, ve které se nejčastěji se objevující znaky nachází na začátku. Zároveň zakódovaný řetězec obsahuje nízké hodnoty jelikož často se vyskytující znaky jsou na začátku abecedy.

Implementováno jako knihovna `blocks/mtf`.

Run-length kódování

Toto kódování efektivně zkracuje délku kódované zprávy a to tím způsobem, že pokud jsou některé znaky za sebou opakovány jsou nahrazeny čítačem výskytu. Tedy je provedena transformace:

ABC ➡ ABC
ABBC ➡ ABB0C
ABBBBC ➡ ABB2C
ABBBBBBBC ➡ ABB4C

Tedy už v případě 3 výskytů po sobě stejných symbolů dochází ke kompresi zprávy. Navíc v souvislosti s použitím předchozího Run-length kódování je větší pravděpodobnost stejných znaků za sebou, jelikož častější znaky jsou alternovány s větší pravděpodobností.

Implementaci Run-length lze nalézt v `blocks/rle`.

Entropické kódování – Huffmanovo kódování

Přestože téma bylo pojmenované po první transformaci, entropické kódování bylo jeho zdaleka nejsložitější částí. Na výběr je dle zadání z několika možností.
Aritmetické kódování jsem vyzkoušel jako první, ale jelikož jsem nebyl schopen správně spočítat počet bitů nutných pro dostatečnou přesnost fixed-point pravděpodobností jednotlivých znaků, od varianty jsem upustil. Tuto přesnost bylo třeba spočítat tak, aby s rezervou pokryla všechna pravděpodobností rozložení pro danou velikost bloku, který již je zakódován v předešlých krocích. Oproti tomu u Huffmanova kódování toto nehrozí. To spočívá v sestavení binárního vyhledávacího stromu podle frekvence/počtu výskytů jednotlivých znaků v kódované zprávě. A jelikož znaků je pouze 256 je velikost stromu relativně omezená. Nutno dodat, že pro Huffmanovo kódování se vytváří b-strom slučováním podstromů a vzniká tak nevyvážený strom, který přiřazuje kratší kódy frekventovaným znakům. Tedy nelze se spoléhat, že je možné si vystačit s 8 bity na znak (to by

stačilo u vyváženého stromu). Zvolil jsem tedy maximálně 16 bitů na kód pro znak. Podle provedených experimentů se to jeví jako dostatečné.

Průběh kódování lze demonstrovat následovně:

```
Spočti počet výskytů jednotlivých znaků;
Seřaď znaky podle počtu výskytů (od nejmenšího);
Opakuj, dokud není pouze jeden strom:
      První dva znaky v seznamu sluč do stromu a zařaď podle součtu výskytů
Ohodnoť jednotlivé listy stromu Huffmanovým kódem
Ulož tabulku frekvencí znaků jako: 1B znak 2B počet výskytů
Vlož oddělovač
Pro každý znak zprávy:
      Nalezni uzel stromu a ulož jednotlivé bity Huffmanova kódu
Vlož Huffmanův kód pro znak konce zprávy
Vlož oddělovač
```

Samotná implementace binárního vyhledávacího stromu je provedena odděleně v knihovně `blocks/btree`. Huffmanovo kódování lze nalézt v knihovně `blocks/shf`.

Závěr

Implementována bylo kódování a komprese podle blokového schématu ze zadání. Nejobtížnější částí byla bitová manipulace při Huffmanově kódování a správné čtení zakódované zprávy. Taktéž obtížné bylo nalézt rovnováhu mezi velikostí bloku kódované zprávy a velikostí Huffmanova kódu. Tedy moderovat vznik binárního stromu, tak aby nepřesáhl určitou hloubku a zároveň aby nebylo nutné příliš často vkládat seznam použitých znaků a jejich frekvenci.
U souborů malé velikosti nemá tato implementace komprimační charakter. Naopak, z důvodů ukládání indexu pro BWT a vkládání hlavičky u Huffmanova kódování naopak velikost zakódovaného textu narůstá. Plný potenciál tohoto řešení se projevuje až u souborů větší velikosti (pohybující se od tisíců znaků výše). Zvolené řešení však není z nejrychlejších, jako příklad uvádím test na přiloženém testovacím souboru:

Čas kódování	119,07s
Poměr komprese	Velikost zmenšena o 51% (379 147B komprimovaný soubor, 768 711B originál)
Čas dekódování	7,45s

Jak je vidět, čas kódování je neproporcionálně dlouhý. To může být způsobeno například naivní implementací Burrows-Wheelova kódování, kdy implementované řešení tvoří celé permutační řetězce místo indexování do stávajícího.