IDATT2101 Øving 6

Komprimering og dekomprimering

Trang Minh Duong

Denne oppgaven har jeg valgt å benytte en aritmetisk rangekoder, Burrows-Wheeler transformasjon (BWT) og Lempel-Ziv predikering (LZP).
Aritmetisk koding deler et intervall proporsjonalt med symbolfordelingen. I vår kvotient-baserte variant (symmetrisk mellom koder og dekoder) gjelder for et symbol med kumulative grenser $[L, H)$ over total T:

$$q = \left\lfloor \frac{\text{range}}{T} \right\rfloor, \qquad \text{low} \leftarrow \text{low} + q \cdot L, \qquad \text{range} \leftarrow q \cdot (H - L).$$

Når range $< 2^{24}$ normaliserer vi ved å sende ut én byte (via `shift_low()`) og venstreforskyve `range` og `low`. For å håndtere potensielle carry uten backtracking bruker vi en cache-byte og en teller for utsatte `0xFF`-bytes (`ff_count`). Avslutning skjer ved å biasere `low` inn i midten av den aktive byten (`low += 0x01000000`) og flushe nøyaktig fire bytes, dekoderen vil da konsumere eksakt nyttelasten.

```
1  // Fenwick tre for kumulative frekvenser
2  struct Fenwick {
3      int n;
4      vector<uint32_t> bit;
5      uint64_t total;
6      Fenwick(int n_ = 257): n(n_), bit(n_+1 ,0), total(0) {}
7
8      void add(int idx, uint32_t delta) {
9          for(int i = idx + 1; i <= n; i += i & -i) {
10             bit[i] += delta;
11         }
12         total += delta;
13     }
14     uint64_t sumPrefix(int idx) const {
15         uint64_t s = 0;
16         for(int i = idx + 1; i > 0; i -= i& - i) s += bit[i
               ];
17         return s;
18     }
19     int findByCumulative(uint32_t off) const {
20         int lo = -1, hi = n - 1;
21         while (hi - lo > 1) {
22             int mid = lo + ((hi - lo) >> 1);
23             if (sumPrefix(mid) <= off) lo = mid;
24             else hi = mid;
25         }
26         return hi;
27     }
28 };
```

Listing 1: Fenwick-tre

Vi bruker en Fenwick-tre (BIT) over alfabetet $\sigma = 257$ for kumulative frekvenser. Dette gir $O(\log \sigma)$ oppslag/oppdatering og effektiv reskalering. Når totalsummen overstiger 2^{20} halveres frekvensene (avrundet oppover) og treet bygges på nytt. Reskalering unngår overflow og gjør at modellen "glemmer" gamle data.

BWT permuterer strengen slik at like kontekster samles, noe som typisk gir lengre løp (runs) og enklere modellering. Vi konstruerer en sirkulær suffiks-tabell ved å rangere parene $(\text{rank}[i], \text{rank}[i+k])$ og bruke stabil counting sort for hver dobling $k \leftarrow 2k$. Dette gir i praksis nær lineær tid på tekst, med teoretisk $O(n \log n)$. **Primærindeks** `primary` er posisjonen til originalstrengen i den sorterte, sirkulære matrisen og trengs for entydig invers. **Invers via LF-mapping** lar L være siste kolonne og F første kolonne av BWT-matrisen. Med opptelling av tegn og startoffset `first[]`, bygger vi

$$\text{next}[i] = \text{first}[L_i] + \text{tally}_{L_i}^{(\text{før } i)},$$

og henter originalen baklengs ved å starte i `primary` og iterere $j \leftarrow \text{next}[j]$. Dette er $O(n)$.

```
1  static void bwt_inverse(const vector<uint8_t>& bwt, uint32_t
       primary, vector<uint8_t>& out) {
2      const int n = (int)bwt.size();
3      out.resize(n);
4      if (primary >= (uint32_t)n) throw runtime_error("BWT
           primary out of range");
5
6      array<uint32_t, 256> cnt{};
7      for (uint8_t c : bwt) cnt[c]++;
8
9      array<uint32_t, 256> first{};
10     int sum = 0;
11     for (int c = 0; c < 256; ++c) { first[c] = sum; sum +=
           cnt[c]; }
12
13     // LF mapping
14     vector<uint32_t> tally(256, 0), next(n);
15     for (int i = 0; i < n; ++i) {
16         uint8_t c = bwt[i];
17         next[i] = first[c] + tally[c]++;
18     }
19
20     int j = (int)primary;
21     for (int i = n - 1; i >= 0; --i) {
22         out[i] = bwt[j];
23         j = next[j];
24     }
25 }
```

Listing 2: BWT invers

SA-konstruksjon krever $O(n)$ minne, derfor brukes blokker på $\sim$1 MiB for å balansere

tid, minne og kompresjon. (Se bruk av `primary` og SA i listing 3)

LZP forsøker å forutsi neste byte ut fra en kort kontekst. Vi bruker en 4-byte kontekst $(c_{t-4}, \ldots, c_{t-1})$ som hashes (Murmur-inspirert miks) til en tabell med 2^{20} oppføringer ($\approx$1 MiB). Tabellen lagrer forrige observerte byte for konteksten. Hvis prediksjonen matcher den faktiske byten koder vi ett spesialsymbol (256), ellers koder vi literal $0 \ldots 255$ og oppdaterer tabellen. **Hashkollisjoner** gir bare tapt prediksjon (en literal sendes), men er ikke farlige for korrekthet. I startfasen ("kald tabell") ser man typisk flere litteraler og treffraten øker etter hvert. (Se bruk av MurmurHash3 i listing 5)

Samspill med BWT og rangekoder-en: BWT produserer lokalt like kontekster, som øker LZP-treff. Range coderen koder så symbolsekvensen (256 = treff, 0–255 = literal) nær entropi-optimalt. Valg av paramentrene er på grunn av at kontekstlengde 4 og tabellstørrelse 2^{20} er et kompromiss mellom minne og kvalitet.

Kompleksitet

Range coder har lineær tid $O(n)$, siden hvert symbol kodes med noen få multiplikasjoner og skift.

```
1  //RangeEncoder
2  ...
3  inline void encode(uint32_t lo, uint32_t hi, uint32_t total)
       {
4          uint32_t freq = hi - lo;
5          if (!freq) { ++hi; ++freq; }
6
7          // quotient-based split
8          uint32_t q = range / total;
9          low   += (uint64_t)q * lo;
10         range  = (uint32_t)((uint64_t)q * freq);
11
12         normalize();
13     }
14
15 //RangeDecoder
16 ...
17 inline uint32_t get_count(uint32_t total) const {
18         uint32_t q = range / total;
19         uint32_t t = code / q;
20         if (t >= total) t = total - 1;
21         return t;
22     }
23
24     inline void remove_symbol(uint32_t lo, uint32_t hi,
           uint32_t total) {
25         uint32_t q = range / total;
26         code  -= (uint32_t)((uint64_t)q * lo);
27         range  = (uint32_t)((uint64_t)q * (hi - lo));
28         normalize();
29     }
```

Listing 3: deler av RangeEncoder og RangeDecoder

For **BWT**, sortering av sirkulære suffikser har en teoretisk kompleksitet på $O(n \log n)$, men implementasjonen min bruker en modifisert counting sort som i praksis gir nesten lineær tid for tekstfiler.

```cpp
// BWT & bruk av sirkular suffiks array
static vector<int> build_sa(const vector<uint8_t>& s) {
    const int n = (int)s.size();
    if (n == 0) return {};
    vector<int> sa(n), r(n), nr(n), key(n);

    for (int i = 0; i < n; ++i) { sa[i] = i; r[i] = (int)s[i]; }

    auto counting_sort = [&](const vector<int>& key, int max_key, vector<int>& idx) {
        vector<int> cnt(max(256, max_key + 2), 0);
        for (int i = 0; i < n; ++i) ++cnt[key[idx[i]] + 1];
        for (int i = 1; i < (int)cnt.size(); ++i) cnt[i] += cnt[i - 1];
        vector<int> out(n);
        for (int i = 0; i < n; ++i) out[cnt[key[idx[i]]]++] = idx[i];
        idx.swap(out);
    };

    for (int k = 1; k < n; k <<= 1) {
        for (int pos = 0; pos < n; ++pos) key[pos] = r[(pos + k) % n];
        counting_sort(key, *max_element(key.begin(), key.end()), sa);

        for (int pos = 0; pos < n; ++pos) key[pos] = r[pos];
        counting_sort(key, *max_element(key.begin(), key.end()), sa);

        // re-rank
        nr[sa[0]] = 0;
        int distinct = 1;
        for (int i = 1; i < n; ++i) {
            int a = sa[i - 1], b = sa[i];
            if (r[a] != r[b] || r[(a + k) % n] != r[(b + k) % n]) ++distinct;
            nr[b] = distinct - 1;
        }
        r.swap(nr);
        if (distinct == n) break; // all unique
    }
    return sa;
}

static void bwt_transform(const vector<uint8_t>& src, vector<uint8_t>& dst, uint32_t& primary) {
    const int n = (int)src.size();
    dst.resize(n);
    auto sa = build_sa(src); // cyclic order
    primary = 0;
    for (int i = 0; i < n; ++i) {
        int j = sa[i];
        dst[i] = src[(j + n - 1) % n];
        if (j == 0) primary = i;
    }
}
```

Listing 4: BWT & bruk av sirkular suffiks array

LZP har en lineær tid $O(n)$ med konstant minnebruk per tegn, basert på et hashoppslag av de fire siste tegnene (MurmurHash3).

```
1  struct LZP {
2      static constexpr uint32_t HASH_SIZE = 1<<20;
3      static constexpr uint32_t HASH_MASK = HASH_SIZE - 1;
4      vector<uint8_t> table;
5      LZP(): table(HASH_SIZE, 0) {}
6      static inline uint32_t hash4(uint8_t a, uint8_t b,
           uint8_t c, uint8_t d) {
7          uint32_t x = ((uint32_t)a<<24)|((uint32_t)b<<16)|((
               uint32_t)c<<8)|d;
8          // MurmurHash3 konstanter
9          x ^= x>>17;
10         x *= 0x85ebca6b;
11         x ^= x>>13;
12         x *= 0xc2b2ae35;
13         x ^= x>>16;
14         return x & HASH_MASK;
15     }
```

Listing 5: LZP og MurmurHash

Totalt gir hele komprimeringskjeden $O(n \log n)$ tid og $O(n)$ plass.

Resultat

Fil	Størrelse (før komp)	Størrelse (etter komp)	Forhold (m/ original)
diverse.txt	17 kB	8.38 kB	47.23%
diverse.lyx	179 kB	19.7 kB	11.01%
opg6-kompr.lyx	34 kB	14.5 kB	42.65%
Jules Verne	595 kB	270 kB	45.38%
enwik8.txt	100 MB	45.7 MB	45.70%

Tabell 1: Tidsmåling for allerede sortert liste med unike n og forholdene

```
$ rm -f ./oeving6.exe t.bsc t.out
g++ -O3 -std=gnu++17 -Wall -Wextra -o oeving6.exe oeving6.cpp

./oeving6.exe c "diverse.txt" "diverse_txt.bsc"
ls -l "diverse_txt.bsc"
./oeving6.exe d "diverse_txt.bsc" "diverse_txt.out"
cmp -s "diverse.txt" "diverse_txt.out" && echo "OK diverse.txt" || echo "DIFF diverse.txt"

./oeving6.exe c "diverse.lyx" "diverse_lyx.bsc"
ls -l "diverse_lyx.bsc"
./oeving6.exe d "diverse_lyx.bsc" "diverse_lyx.out"
cmp -s "diverse.lyx" "diverse_lyx.out" && echo "OK diverse.lyx" || echo "DIFF diverse.lyx"

rm -f "oeving6.exe" "enwik8.bsc" "enwik8.out"
g++ -O3 -std=gnu++17 -Wall -Wextra -o oeving6.exe oeving6.cpp

./oeving6.exe c "enwik8.txt" "enwik8.bsc"
./oeving6.exe d "enwik8.bsc" "enwik8.out"
cmp -s "enwik8.txt" "enwik8.out" && echo "OK enwik8" || echo "DIFF enwik8"

./oeving6.exe c "Twenty_thousand_leagues_under_the_sea.txt" "Twenty_thousand_leagues_under_the_sea.bsc"
./oeving6.exe d "Twenty_thousand_leagues_under_the_sea.bsc" "Twenty_thousand_leagues_under_the_sea.out"
cmp -s "Twenty_thousand_leagues_under_the_sea.txt" "Twenty_thousand_leagues_under_the_sea.out" && echo "OK verne" || echo "DIFF verne"

./oeving6.exe c "opg6-kompr.lyx" "opg6-kompr.bsc"
./oeving6.exe d "opg6-kompr.bsc" "opg6-kompr.out"
cmp -s "opg6-kompr.lyx" "opg6-kompr.out" && echo "OK kompr" || echo "DIFF kompr"
```

Figur 1: Bash sammenlign to filer

```
[INFO] input bytes = 17060
[DBG] payload bytes = 8555
-rw-r--r-- 1 duong duong 8583 Nov  3 16:54 diverse_txt.bsc
OK diverse.txt
[INFO] input bytes = 182355
[DBG] payload bytes = 20156
-rw-r--r-- 1 duong duong 20184 Nov  3 16:54 diverse_lyx.bsc
OK diverse.lyx
```

(a) Log 1 (diverse.txt/.lyx)

```
[DBG] payload bytes = 496732
[DBG] payload bytes = 495324
[DBG] payload bytes = 192608
OK enwik8
[INFO] input bytes = 609116
[DBG] payload bytes = 277141
OK verne
[INFO] input bytes = 34432
[DBG] payload bytes = 14851
OK kompr
```

(b) Log 2 (enwik8, verne, kompr)

Figur 2: Log for kjøringen

Begrensninger

Programmet og filformatet bruker `uint64_t` for filstørrelser og intern `low`-akkumulator i rangekoderen. Jeg bygger programmet som 64-bit (`ucrt64` / `-m64`). På 32-bit vil det fortsatt kompilere, men kan få ytelsesstraff og strengere grenser på adressérbart minne. En potensiel begrensing er filstørreslen. Total lengde i header lagres som 64-bit ($\leq 2^{64} - 1$), men hver blokk lagrer komprimert lengde som 32-bit (< 4 GiB per blokk). Med blokkstørrelse 1 MiB er dette uansett uproblematisk.

Presisjon i rangekoder er også en annen begrensing. Intervallbredden (`range`) er 32-bit (vanlig i praksis), `low` er 64-bit for å håndtere carry/underflow trygt. Svært store totalsummer reskaleres (2^{20}) for å unngå overflow i modellen. Angående minnebruket har LZP-tabellen 2^{20} oppføringer (≈ 1 MiB). I tillegg kommer arbeidsbuffere for BWT og blokker ($O(n)$ per blokk).

Resultatet varierer også basert på datasettene og mønstret deres. Kjeden (BWT+LZP+RC) fungerer best på data med repetisjon (tekst, noen binærfiler). Helt entropirike data gir liten/ingen komprimering. Lese-/skrivefunksjonene bruker eksplisitt little-endian, så formatet er portabelt mellom plattformer, gitt 8-bit `char`.