Practicum Algoritmen en Datastructuren

– voorjaar 2011 –

Opgave 4: Datacompressie

1 Achtergrond

Het doel van *datacompressie* is om de hoeveelheid ruimte, die wordt gebruikt om gegevens op te slaan, te verminderen. Het idee is dat 'overbodige' informatie niet wordt opgeslagen, terwijl het toch mogelijk blijft het origineel te reconstrueren. Soms accepteert men hierbij een (gering) verlies aan informatie, bijvoorbeeld bij comprimeren van muziek (MP3), plaatjes (JPEG) en video (MPEG). Bij overbodige informatie moet men dan denken aan geluiden die je toch niet hoort, of kleurnuances die je (waarschijnlijk) toch niet ziet. Dit verlies gaat dan ten koste van de kwaliteit van het gedecomprimeerde plaatje of muzieknummer, maar levert wel een enorme besparing aan ruimte op. Dit zijn de zogenaamde 'lossy' compressiealgoritmes.

Hoewel je soms een tekst nog wel zou kunnen reproduceren als er letters zijn weggevallen, kan dit niet de bedoeling zijn als je tekstbestanden comprimeert. Hiervoor zijn er 'lossless' compressiealgoritmes die, zoals de naam al aangeeft, geen informatie verliezen. Je kunt tekstbestanden redelijk comprimeren door bijvoorbeeld veel voorkomende (lange) woorden af te korten. Het is niet nodig om iedere keer voor ieder woord alle letters van dat woord uit te schrijven. Zo zouden we een tekstbestand ook kunnen vervangen door een woordenlijst en een reeks verwijzingen (codes) naar die woordenlijst. Mits er een redelijk aantal woorden meerdere keren gebruikt worden, zal dit minder ruimte vergen dan de originele tekst. Met behulp van die woordenlijst kunnen we het bestand decomprimeren tot de oorspronkelijke tekst.

In deze opgave ga je een bekend datacompressie–algoritme implementeren dat geen informatie verliest: *Lempel-Ziv-Welch*, of kortweg *LZW-compressie*. Dit algoritme wordt bijvoorbeeld gebruikt in de GIF, TIFF en ZIP bestandsformaten. Ondertussen is het patent op LZW verlopen en kun je weer rustig zelf een implementatie maken zonder aangeklaagd te worden door Unisys.

2 Leerdoelen

Het belangrijkste leerdoel van deze opgave is het vertrouwd raken met het werken met (zoek)bomen. Na afloop van deze opdracht ben je in staat om:

- (zoek)bomen te gebruiken als onderliggende implementatie van andere datastructuren en als datastructuur voor een algoritme;
- datacompressie, met name LZW, te beschrijven en te implementeren met behulp van geordende boomstructuren;

3 Instructie

Bestudeer allereerst de onderdelen uit hoofdstuk 17 van het dictaat en/of hoofdstuk 12 van het boek die op college zijn besproken, de sheets over dit onderwerp zoals gebruikt tijdens het college (bekijk beslist ook de programma-voorbeelden) en uiteraard je op het hoorcollege gemaakte aanvullende eigen aantekeningen. Begin pas daarna aan de volgende deelopdrachten. Zorg ervoor, dat je (op papier) al een ontwerp van je LZW implementatie hebt gemaakt, alvorens je aanschuift achter de pc om je code in te voeren en uit te testen. Voor deze opgave heb je twee weken de tijd.

4 Probleemschets

LZW-compressie is gebaseerd op het volgende idee: stel je komt tijdens het comprimeren van een tekst een bepaalde deeltekst (reeks van bytes) voor het eerst tegen. Dan is de kans redelijk dat diezelfde deeltekst verderop in de tekst nog eens voorkomt. We geven die deeltekst bij z'n eerste voorkomen een (unieke) code en bij ieder volgend voorkomen gebruiken we die code in plaats van de deeltekst zelf. Bij het decomprimeren vervangen we iedere code weer door de bijbehorende deeltekst.

Compressie (Lempel-Ziv)

Tijdens het comprimeren bepalen we steeds de *langste deeltekst* aan het begin van de nog te comprimeren tekst die we al eerder aan een code hebben gekoppeld (en dus al eerder zijn tegengekomen) en vervangen deze door die code. Uitgebreid met het eerstvolgende teken vormt de gevonden deeltekst een nieuwe deeltekst waarvoor we nog geen code hebben, anders zou deze immers de langste deeltekst zijn. Aan deze nieuwe deeltekst wijzen we een nieuwe code toe, zodat we die code kunnen gebruiken als we deze nieuwe (langere) substring later weer tegenkomen. De deeltekst en de bijbehorende code worden in een tabel opgeslagen. Dit herhalen we tot dat de hele invoer gecomprimeerd is.

We kiezen voor deze implementatie van het LZW-algoritme een vaste grootte voor alle codes, te weten 12 bits. Dat betekent dat we in totaal $2^{12} = 4096$ verschillende codes kunnen gebruiken.

Tijdens het coderen hebben we dus een administratie die deelteksten en codes met elkaar relateert: de `codetabel`. Op ieder moment is er een deeltekst `deeltekst` van gelezen tekens waarvoor nog geen uitvoer is gegenereerd. Na het lezen van het eerstvolgende teken uit de invoer (aangeduid met het `volgend_teken`) zijn er twee mogelijkheden:

1. De codetabel bevat al een code voor de huidige deeltekst uitgebreid met het volgende teken. Er hoeft nu (nog) geen uitvoer geproduceerd te worden. Het volgende teken wordt toegevoegd aan de deeltekst.

2. De codetabel bevat nog geen code voor de huidige deeltekst plus het volgende teken. Nu wordt de code behorend bij de huidige deeltekst naar de uitvoer gestuurd. Bovendien wordt er een nieuwe code toegekend aan de combinatie van de deeltekst plus volgende teken. Dit laatste gebeurt overigens alleen indien er nog ongebruikte codes zijn (oftewel de grootte van de codetabel is kleiner dan 4096). De nieuwe deeltekst bestaat vervolgens enkel uit het huidige teken.

In pseudo-code:

```
string deeltekst = lees volgend teken;
while (er is nog invoer)
{   char volgend_teken = lees volgend teken;
    if (komtVoor (deeltekst + volgend_teken, codetabel))
        deeltekst = deeltekst + volgend_teken;
    else
    {   schrijf de code voor deeltekst weg;
        if (er zijn nog ongebruikte codes)
           voegToe (deeltekst + volgend_teken, codetabel);
        deeltekst = volgend_teken;
    }
}
schrijf de code voor deeltekst weg;
```

Merk op aan het begin van iedere iteratie van de while-loop moet gelden dat de deeltekst van dat moment in de tabel voorkomt, m.a.w. `komtVoor` (`deeltekst`, `codetabel`) is een *invariant* van de while-loop.

Iedere deeltekst die slechts uit één enkel teken bestaat behandelen we speciaal door hier al op voorhand de ASCII-codes van dat teken aan toe te wijzen. Gevolg hiervan is dat de codes 0 tot en met 255 al zijn uitgereikt voordat de compressie gestart is. Op deze manier zorgen we ervoor dat de invariant van de while-loop altijd geldt.

In het volgende voorbeeld werken we niet met alle 256 mogelijke letters maar met een alfabet dat slechts uit vier letters bestaat: $\{a, b, c, d\}$. Hierin hebben de deelteksten a tot en met d de codes 0 tot en met 3. Voor de invoer "*abacababa*" krijgen we dan:

invoer	`deeltekst`	`volgend_teken`	`codetabel`	uitvoer
abacababa			a,b,c,d	
bacababa	a		a,b,c,d	
acababa	a	b	a,b,c,d	
acababa	a	b	a,b,c,d,ab	0
cababa	b	a	a,b,c,d,ab,ba	1
ababa	a	c	a,b,c,d,ab,ba,ac	0
baba	c	a	a,b,c,d,ab,ba,ac,ca	2
aba	a	b	a,b,c,d,ab,ba,ac,ca	
ba	ab	a	a,b,c,d,ab,ba,ac,ca	4
a	a	b	a,b,c,d,ab,ba,ac,ca,aba	
	ab	a	a,b,c,d,ab,ba,ac,ca,aba	
	aba		a,b,c,d,ab,ba,ac,ca,aba	8

Merk op dat de eerste iteratie van de while-loop begint bij regel 3. Verder hebben we de codes die horen bij de deelteksten in de codetabel weggelaten: ze komen immers overeen met de positie van de betreffende tekst in de tabel (b.v. de deeltekst *ba* heeft code 5). De uitvoer voor de gegeven tekst is dus $0, 1, 0, 2, 4, 8$. Voor een volledig alfabet van 256 verschillende tekens (bytes) zou de uitvoer $97, 98, 97, 99, 256, 260$ zijn.

Efficiënte bomen voor substrings (Welch)

Indien we de codetabel inderdaad zoals het voorbeeld suggereert als een rij van teksten zouden representeren dan zou dit leiden tot een nogal inefficiënte implementatie: bij iedere nieuwe deeltekst moet de complete tabel worden doorlopen om na te gaan of de deeltekst al eerder is aangetroffen. Voor elke vergelijkingsstap heb je de (dure) stringvergelijking nodig.

Om dit zoeken naar deelteksten efficiënter te kunnen doen gebruiken we geen tabel maar bouwen we een zogenaamde *prefix-boom* op: een speciale *zoekboom* waarmee we voor een gegeven invoertekst snel kunnen beslissen wat de maximale prefix is die al eerder is aangetroffen en welke code daar toen aan is toegekend.

Iedere knoop van die zoekboom bevat daartoe een *code*, een *teken*, en *drie onderbomen*. Het is de bedoeling dat we met behulp van deze boom alle deelteksten representeren die we gedurende het compressieproces zijn tegengekomen. Je vindt een bepaalde deeltekst door, beginnende bij de wortel, af te dalen en steeds het in de boom opgeslagen teken te vergelijken met het huidige teken in je tekst. Stel bijvoorbeeld dat je wil nagaan welke prefix van tekst S al in boom R is opgeslagen. We nemen even aan dat je bij het i^e teken van S, zeg S_i, bent aanbeland (dat wil zeggen dat je de eerste i tekens van S al hebt herkend) en dat je in de zoekboom bij deelboom T zit. Er zijn nu 2 mogelijkheden:

1. T is leeg. In dat geval vormen de eerste i tekens van S de langste prefix die al was opgeslagen. De prefix boom kan nu worden uitgebreid door de lege boom te vervangen door een knoop met als letter S_i en code de eerstvolgende nog niet gebruikte code.

2. T is niet leeg. Nu wordt het teken van de wortel van T, zeg T_c, vergeleken met S_i. We kunnen 3 gevallen onderscheiden:

   $T_c = S_i$ Dit betekent dat ook het i^e teken van S voorkomt en dus de prefix met één teken kan worden uitgebreid. i kan derhalve worden opgehoogd en het zoekproces gaat verder met T.`volgende` als deelboom.

   $T_c < S_i$ Je weet nu (nog) niet of teken S_i voorkomt maar maakt gebruik van de ordening van de zoekboom door verder te gaan met zoeken in de onderboom T.`groter`.

   $T_c > S_i$ Idem, maar nu ga je verder in T.`kleiner`.

Zo'n zoekboom voor LZW kun je representeren met onderstaande klasse (waarin de tekens niet als **`char`**, maar als **`int`** worden opgeslagen. Dit blijkt in de praktijk net ietsje makkelijker te zijn):

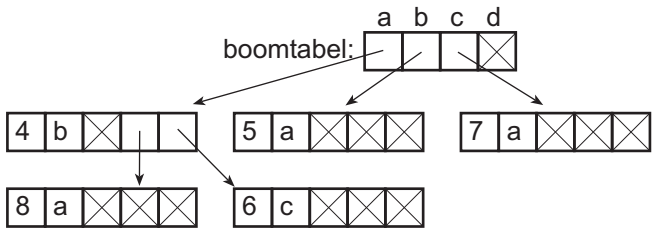
```
private static class LZWNode
{
    private int node_char, node_code;

    private LZWNode smaller, longer, greater;

    private LZWNode (int code, int c)
    {
        node_code = code;
        node_char = c;
        smaller = longer = greater = null;
    }

}
```

De datastructuur lijkt erg op een gewone geordende boom: er zijn onderbomen voor substrings die beginnen met een karakter dat alfabetisch vóór (`smaller`) en ná (`greater`) het karakter van de huidige knoop komen. Daar 'doorheen' worden substrings opgeslagen als ketens (`longer`) van karakters. De exacte huidige substring heb je in de implementatie niet nodig, je hoeft slechts te weten welke code hoort bij de langste prefix van de deeltekst die voorkomt in de boom. De huidige substring wordt dus impliciet bepaald door de positie van de knoop waar we ons op dat moment bevinden binnen de totale zoekboom. Het pad van de wortel van de boom tot die knoop vormt namelijk de huidige substring. Ga dit zelf na!

Een optimalisatie die we gebruiken is om, in plaats van een zoekboom te bouwen voor de 256 vaste codes, een rij van bomen te gebruiken: de `boomtabel`. Deze keuze heeft als voordeel dat we het eerste teken van een nieuwe substring als index kunnen gebruiken waarmee meteen de zoekboom kunnen selecteren die overeenkomt met alle voortzettingen van dat teken. Voor het bovenstaande voorbeeld met slechts vier letters zal de zoekboom er aan het eind als volgt uitzien: Zoals gezegd, bevat deze boom alleen de voortzettingen die uit meer dan 1 tekens bestaan.

Deelopdracht 1

Implementeer een programma dat bestanden kan comprimeren op de hierboven beschreven wijze. Test je programma eerst met wat simpele voorbeeldjes als "*abacababa*" en "*aaaaaaaaaa*". Als dat goed werkt probeer je grotere dingen zoals één van je `.java`-files of een ander (klein) tekstbestand.



Decompressie

Een van de aardige eigenschappen van dit compressie–algoritme is dat het niet nodig is om een zoekboom/codetabel mee op te slaan. Uit de gegenereerde codes kan de decoder de tabel zelf opnieuw opbouwen. Bij iedere code c die tijdens decompressie wordt ingelezen weten we dat de bijbehorende deeltekst niet alleen moet worden weggeschreven (als resultaat van de decompressie van c) maar ook dat deze deeltekst de *prefix* vormt van de eerstvolgende nieuwe code in de tabel. De complete deeltekst vind je door deze prefix te combineren met het eerste teken van de deeltekst die hoort bij de code die direct na c wordt ingelezen. Om dit administratief geregeld te krijgen houd je tijdens decompressie de laatst ingelezen code bij (`vorige_code`) en de eerstvolgende nieuwe code (`volgende_code`) in de codetabel.

Dit leidt tot het volgende algoritme in pseudo-Java.

```
int volgende_code = 256, vorige_code = lees code;
schrijf vorige_code weg;
while (er is nog invoer)
{   int nieuwe_code = lees code;
    if (er zijn nog ongebruikte codes)
    {   String prefix = haalUitTabel(codetabel,vorige_code);
        voegPrefixToe (codetabel, prefix, volgende_code);
        voegTekenToe (codetabel, eersteTeken(codetabel,nieuwe_code), volgende_code);
    }
    schrijf haalUitTabel(codetabel,nieuwe_code) weg;
}
```

Omdat er nu deelteksten bij codes gezocht moeten worden (in plaats van codes bij deelteksten), gebruiken we een andere, veel eenvoudigere datastructuur. Per code houden we bij waar (van welke eerder tegengekomen deeltekst) het een extensie van is. Daarnaast slaan we het teken dat toegevoegd wordt aan deze deeltekst op. Ga na hoe decompressie van het boven gegeven voorbeeldje werkt. De datastructuur zou je als volgt kunnen implementeren in Java:

```
private class CodeRepr
{   int prefix, last_char; };

static final int MaxAantalCodes = 4096;

private CodeRepr decompressieTabel [];

    ...
    decompressieTabel = new CodeRepr [MaxAantalCodes]
    ...
```

De implementatie van decompressie blijkt eenvoudiger te zijn dan die voor het compressiedeel. Het enige probleem treedt op wanneer er tijdens compressie een code gebruikt wordt onmiddellijk nadat hij aan de compressietabel is toegevoegd. Dit gebeurt bijvoorbeeld als we de string

"*aaaaaaaaaa*" comprimeren. We moeten nu niet meteen proberen om tijdens decompressie eerst de deeltekst op te halen en weg te schrijven en dan pas de tabel aan te passen, maar eerst de tabel aanpassen en dan pas de gedecomprimeerde deeltekst bepalen en wegschrijven. Je ziet dat terug in de voorbeeldcode. Verder is het essentieel dat je eerst de prefix in de tabel zet en daarna pas het extra teken. Probeer zelf na te gaan wat er gebeurt als je de volgorde van deze acties wijzigt wanneer je "*aaaaaaaaaa*" na compressie weer decomprimeert.

Uitvoer

Als we naïef codes naar een uitvoerbestand schrijven hebben we minstens 2 bytes nodig voor iedere code. We hebben dan een overhead van 4 bits (33%) per code. Voor een compressie–algoritme kan dit nooit de bedoeling zijn. Verbeter het algoritme door twee opeenvolgende codes in 3 bytes te zetten alvorens ze weg te schrijven. Nog efficiënter is het om gebruik te maken van een variabel aantal bits, maar dat hoeven jullie niet te implementeren.

Deelopdracht 2

Implementeer een programma dat bestanden kan comprimeren en decomprimeren op de hierboven beschreven wijze. Je implementatie dient ook 2 codes in 3 bytes weg te schrijven. Test je programma opnieuw met wat simpele voorbeeldjes als "*abacababa*" en "*aaaaaaaaaa*". Als dat goed werkt probeer je grotere dingen zoals je Java-programma of een willekeurige executable ergens op de harde schijf van je PC. Bekijk hoeveel compressie in deze gevallen oplevert. Neem een kort overzicht hiervan op als commentaar in je programma.

5 Producten

Als producten heb je je uitgeteste Java-code, eerst alleen met een implementatie voor LZW-compressie, daarna met zowel compressie als decompressie met 2 codes in 3 bytes. Zorg ervoor dat je programma voldoet aan de gevraagde specificaties en aan de kwaliteitscriteria zoals die gesteld worden.

6 Inleveren van je producten

Je krijgt voor deze opdracht twee weken de tijd. De eerste week werk je aan het compressie-deel (deelopdracht 1), de tweede aan het decompressie deel (deelopdracht 2). In Blackboard worden hiervoor 2 assignments aangemaakt, de eerste met deadline 28 februari, 8:30 uur, de tweede één week laten, 7 maart 8:30 uur. Vergeet niet om in de uitwerking duidelijk jullie namen en studentnummers te vermelden.