1 Algoritmi de compresie

1.1 Range encoding

Ideea de "range encoding" a apărut prima dată în lucrarea lui G.N.N Martin[1] .

Spunând că lăţimea unui mediu de stocare este *s*, sau *d* cifre din baza *b*, înţelegem că poate lua una din cele s valori, sau una din cele b^d valori distincte.

Dacă stocăm o literă, şi restrângem mediul de stocare la una din t valori distincte, atunci lăţimea codării caracterului este s/t, şi lăţimea rămasă este t, în care putem stoca un REST de lăţime *t* . Setul de t valori ce pot reprezenta litera, se numeşte DOMENIUL literei în lăţimea spaţiului de stocare.

De exemplu dacă domeniul unei litere într-un spaţiu de stocare cu lăţimea 256 este [240, 250), atunci lăţimea lilterei este 25.6, şi lăţimea rămasă este 10.

Dacă un domeniu are forma $[B, T)$, atunci îl putem combina cu un rest prin aritmetică simplă. Dacă dorim să stocăm $i \in [0, T - B)$, ca rest pentru $[B, T)$, atunci valoarea stocată este $B + i$; sau dacă $[i, j) \subseteq [0, T - B)$ trebuie stocat ca rest parţial pentru $[B, T)$, atunci valoarea stocată este constrâns la $[B + i, B + j)$.

Fie $f(a)$ probabilitatea ca litera 'a' să apară în orice context dat. Presupunem că alfabetul este ordonat, şi definim $F(a)$ ca fiind probabilitatea unei literi precedente lui 'a' să apară în acelaşi context, adică:

$$F(a) = \sum_{x<a} f(x)$$

În continuare voi nota $f(a)$ cu fa, $F(a)$ cu Fa, $s.fa$ cu sfa.

Shannon a arătat, că pentru minimizarea cifrelor necesare pentru reprezentarea mesajului într-o bază *b*, ar trebui să codăm fiecare literă *'a'*, a.î. lăţimea să fie $-\log_b(fa)$ cifre, adică $1/fa$ în lăţime absolută.

Nu putem realiza acest lucru exact, dar dacă codăm *'a'* într-un spaţiu de stocare cu lăţimea *s*, ca şi $[\lfloor s.Fa \rfloor , \lfloor s(fa + F$ atunci lăţimea literei se aproprie de $1/fa$ pentru $s.fa \gg 1$. Dacă $s.fa \geq 1$, atunci fiecare literă se poate coda, şi decoda fără echivoc.

1.1.1 Decodificare

O literă *'a'*, împreună cu restul său este codificat (într-un spaţiu de stocare de lăţime *s*) ca $i \subseteq [\lfloor sFa \rfloor , \lfloor s(Fa + fa) \rfloor]$. Fie $L(j)$ ultima literă *e* din alfabet pentru care $Fe < j$. Putem folosi *L* pentru a deduce *'a'*, ştiind *i*:

$$\lfloor sFa \rfloor \;\leq\; i < \lfloor s(Fa + fa) \rfloor \quad \Rightarrow sFa < i + 1 \leq s(Fa + fa) \quad \Rightarrow Fa < \frac{i+1}{s} \leq Fa + fa$$

$\Rightarrow a = L\left(\frac{i+1}{s}\right)$

Trebuie ţinut cont şi de erorile de rotunjire la calcularea lui $\frac{i+1}{s}$. Putem verifica dacă litera este corectă prin confirmarea relaţiei $\lfloor sFa \rfloor \leq i < \lfloor s(Fa + fa) \rfloor$.

După ce am dedus *'a'*, restul este $i - \lfloor sFa \rfloor$, şi a fost codat cu o lăţime de $\lfloor s(Fa + fa) \rfloor - \lfloor sFa \rfloor$.

1.1.2 Algoritmul de codificare/decodificare

Dacă o literă *'a'* se codifică ca $[B, T)$, lăţimea rămasă este $T - B$. Dacă acesta e prea mic, îl putem extinde prin adăugarea unei cifre (în baza *b*), domeniul devenind: $[Bb, Tb)$, şi lăţimea rămasă devine $(T-B)b$. La decodificare ignorăm cifra în plus, pentru că codificarea lui *'a'* în lăţimea sb nu este neapărat $[Bb, Tb)$.

Fie $s = b^w$, unde w este numărul (întreg) maxim de cifre în baza *b* pe care îl putem utiliza în mod convenabil.

Codificăm prima literă a mesajului în lăţimea *s*, şi adăugăm atâtea cifre în coadă, cât putem fără să cauzăm ca restul să depăşească lăţimea *s*.

Fie lăţimea spaţiului de stocare după codarea a celei de a *i*-a literă: S_i, de valoare $[B_i, T_i)$; atunci putem coda următoarea literă $A(i + 1)$, in spaţiul de stocare de lăţime $R(i + 1)$, unde:

$$\begin{aligned} R_{i+1} &= (T_i - B_i)b^{k(i+1)} \\ k_{i+1} &= w - \lceil \log_b(T_i - B_i) \rceil \end{aligned}$$

Pentru $i > 0$:

$$\begin{aligned}
[B_i, T_i) &= \left[B_{i-1}b^{k_i} + \lfloor R_i FA_i \rfloor\, , B_{i-1}b^{k_i} + \lfloor R_i(FA_i + fa_i)\rfloor\right) \\
S_i &= \sum_{j=1}^{i} k_j \\
[B_0, T_0) &= [0, 1)
\end{aligned}$$

1.1.3 Exemplu de codificare

Codificarea mesajului: *"NMLNNNKKNML"*

Lăţime rămasă (ajustat)	litera următoare	domeniul literei următoare	Mesaj curent codificat	Domeniul curent mesajului	Lăţime rămasă
1000	N	$[580, 1000)$	N	$[580, 1000)$	420
420	M	$[130, 243)$	NM	$[710, 823)$	113
113	L	$[011, 035)$	NML	$[721, 745)$	24
240	N	$[139, 240)$	NMLN	$[7349, \ldots 450)$	101
101	N	$[058, 101)$	NMLNN	$[7407, \ldots 450)$	43
430	N	$[249, 430)$	NMLNNN	$[74319, \ldots 500)$	181
181	K	$[000, 018)$	NMLNNNK	$[74319, \ldots 337)$	18
180	K	$[000, 018)$	NMLNNNKK	$[743190, \ldots 208)$	18
180	N	$[104, 180)$	NMLNNNKKN	$[7432004, \ldots 080)$	76
760	M	$[235, 440)$	NMLNNNKKNM	$[73420275, \ldots 480)$	205
205	L	$[020, 063)$	NMLNNNKKNML	$[73420295, \ldots 338)$	43

Codul complet trebuie ales cu 7 cifre semnificative (din: $[73420295, 73420338)$), de ex: 7432031.

1.1.4 Implementare algoritm

Se observă că în cazul unui domeniu există 3 zone distincte:

$$\begin{array}{l} [\ \underbrace{13}\ \underbrace{19}\ \underbrace{314}\ \ , \\ \ \ \underbrace{13}_{z1}\ \underbrace{20}_{z2}\ \underbrace{105}_{z3}\ \ ] \end{array}$$

Zona z1 constă din cifre comune tuturor numerelor din domeniu, deci nu vor fi afectate de alegerea restului. Aceste cifre pot fi scrise la ieşire.

Zona_z2 constă din *n* cifre formând un număr db^{n-1}, sau $db^{n-1} - 1$, unde *d* este o singură cifră, şi *b* este baza codificării. În aceste exemplu $n = 2$, şi $d = 2$. Cifrele din această zonă pot fi afectate de alegerea restului, dar care nu sunt necesare pentru a distinge 2 numere din domeniu. Acestea le numim cifre AMÂNATE, şi (d, n) identifică posibilele valori ale cifrelor. Prin convenţie, dacă $n = 0 \quad \Rightarrow d = 0$.

Zona z3 constă din *w* cifre, şi sunt suficiente pentru a distinge între 2 numere din domeniu.

Considerăm domeniul $[B', T']$, cu cifrele transmise: *c*, şi cifrele amânate reprezentate prin (d, n). Fie *x* cifrele transmise după rezolvarea amânării superior:

$$x = cb^n + db^{n-1}$$

atunci putem exprima $[B', T']$, ca: $c, (d, n), [B, T]$, unde $B = B' - xs$, şi $T = T' - xs$. De exemplu $[1319314, 1320105)$ devine $13, (2, 2), [-686, 105)$.

Dacă lăţimea rămasă este $T - B$, şi dacă combinăm $c, (d, n), [B, T]$ cu restul parţial $[i, j) \subseteq [0, T - B]$, atunci creăm domeniul $c, (d, n), [B + i, B + j]$.

Dacă $B+j \leq 0$ atunci putem rezolva cifra amânată inferior, iar dacă $B+i \geq 0$atunci îl putem rezolva superior.

Acest algoritm se poate implementa simplu, fiindcă, dacă domeniul este $c, (d, n), [B, T]$, atunci: $-s < B < T \leq +s$, unde:

d este o singură cifră

n este un întreg mic

c nu trebuie reţinut în codificator/decodificator

Pentru a limita numărul de cifre amânate, putem impune o limită superioară. Putem forţa rezolvarea amânării prin modificarea capetelor domeniului.

Ex:

$$13, (2, 3), [-660, 140] \Rightarrow 13, (2, 3), [-660, 000] \Rightarrow 13199, (0, 0), [340, 1000)$$

$$13, (2, 3), [-140, 660) \Rightarrow 13, (2, 3), [000, 660) \Rightarrow 13200, (0, 0), [000, 660)$$

Prin acesta risipim cel mult 1 bit.

1.2 Modelare statistică

1.2.1 Modelare statistică statică

Una din metodele de determinare a probabilităţii de apariţie a unui simbol este modelarea contextuală finită [6]. Acesta se bazează pe ideea că se calculează probabilităţile pentr un simbol pe baza contextului în care apare. *Contextul* reprezintă simbolii deja întâlniţi. *Ordinul* modelului se referă la numărul de simboluri precedente care alcătuiesc contextul.

Cel mai simplu model cu context finit este un model de ordinul 0. În acest caz probabilităţile unui simbol sunt independente. Pentru implementare este necesar doar un tabel cu frecvenţa de apariţie a simbolurilor.

Pentru un model de ordinul 1 avem nevoie de 256 asemenea tabele, pentru că trebuie să avem contori separaţi pentru fiecare context posibil. Pentru un model de ordinul 2 avem nevoie de 65536 tabele, ş.a.m.d.

O metodă de implementare este de a face 2 parcurgeri asupra datelor: una pentru a determina frecvenţa de apariţie, şi încă una pentru codificarea simbolurilor (folosind un codificator aritmetic, sau un range-encoder).

1.2.2 Modelare statistică adaptivă

Pentru modele de ordinul > 1, spaţiul ocupat de modelul statistic devine foarte mare, în comparaţie cu datele de intrare (şi în multe situaţii le depăşeşte).

Pentru a înlătura acest dezavantaj se evită stocarea modelului. Dar şi decodificatorul trebuie să cunoască modelul, şi acesta nefiind stocat împreună cu datele comprimate, înseamnă că decodificatorul trebuie să-l construiască pas-cu-pas. Acesta se numeşte modelare adaptivă.

În acest caz algoritmii de compresie şi decompresie pornesc cu acelaşi model, codifică simbolul cu modelul curent, şi după aceea reîmprospătează modelul cu noul simbol. Astfel modelul curent se bazează pe caracterele întâlnite deja, cunoscute atât de către programul de compresie, cât şi de către cel de decompresie.

1.3 Compresie folosind coduri distanţă-lungime

În specificaţia formatului DEFLATE [5] (format folosit de Zip) se utilizează coduri distanţă-lungime. O pereche <lungime, distanţă> are următoarea semnificaţie: se copiază începând de la <*poziţia curentă*> - <*distanţă*> la ieşire <*lungime*> octeţi.

De observat, că <*lungime*> poate fi mai mare decât <*distanţă*>. Exemplu: *ABCDX<8,1>*, înseamnă: *ABCDXXXXXXXXX*.

Pentru stocarea acestor coduri, în cazul formatului *DEFLATE* se extinde alfabetul de 256 coduri, cu încă 30 coduri. Codul 256 marchează sfârşitul unui bloc.

Lungimile se reprezintă cu codurile 257-285, împreună cu eventualele biţi extra, vezi tabela 1.

Cod	Biţi	Lungime	Cod	Biţi	Lungime	Cod	Biţi	Lungime
257	0	3	267	1	15,16	277	4	67-82
258	0	4	268	1	17,18	278	4	83-98
259	0	5	269	2	19-22	279	4	99-114
260	0	6	270	2	23-26	280	4	115-130
261	0	7	271	2	27-30	281	5	131-162
262	0	8	272	2	31-34	282	5	163-194
263	0	9	273	3	35-42	283	5	195-226
264	0	10	274	3	43-50	284	5	227-257
265	1	11,12	275	3	51-58	285	0	258
266	1	13,14	276	3	59-66			

Tabela 1: coduri pentru lungime

-

Aceste coduri se generează de obicei prin aplicarea algoritmului LZ77 (Lempel-Ziv 1977[9]).

2 Structuri de date folosite

2.1 Arbori digitali “expanse-based”: arbori Judy

Arborii Judy au fost inventaţi de către Doug Baskins[3], şi implementaţi împreună cu Alan Silverstein[2] la Hewlett Packard. Ulterior algoritmul, şi programul au fost făcute publice.

Un arbore Judy este mai rapid, şi utilizează mai puţină memorie decât alte forme de arbori, cum ar fi: arbori binari, AVL, arbori B, skip-list. Când este folosit ca şi înlocuitor pentru algoritmi de dispersie, este în general mai rapid pentru toate populaţiile.

Judy fiind proiectat ca şi un vector nelimitat, dimensiunea unui vector Judy nu este prealocat, ci creşte, şi descreşte dinamic cu populaţia vectorului.

Judy combină scalabilitatea cu uşurinţa în utilizare. API-ul Judy este accesat prin operaţii simple de inserare, regăsire, şi ştergere. Configurare, şi tuning nu sunt necesare, şi de fapt nici posibile pentru Judy. În plus sortarea, căutarea, numărarea, şi accesul secvenţial sunt incluse în Judy.

Judy poate fi folosit când este nevoie de vectori de mărime dinamică, vectori asociativi. De asemenea Judy poate înlocui multe structuri de date comune, cum ar fi: vectori, vectori sparse, tabele de dispersie, arbori B, arbori binari, liste liniare, skiplists, algoritmit de căutare şi sortare, funcţii de numărare.

O umplere a liniei cache (CPU) înseamnă timp adiţional pentru citire din RAM, când un cuvânt nu este găsit în cache. În calculatoarele actuale acest timp este în zona 50..2000 instrucţiuni. Deci o umplere a liniei cache trebuie evitată când 50 sau mai puţine instrucţiuni pot face acelaşi lucru.

Câteva motive pentru care Judy este mai bun decât arborii binary, arbori B, şi skiplists:

- Judy nu face compromisuri între simplitate şi performanţă/spaţiu (doar API-ul se păstrează simplu)
- Criteriul principal este: Judy e proiectat ca să evite umplerile de linii cache, când e posibil

- Un arbore B necesită o căutare a fiecărui nod, rezultând în mai multe umpleri de linii cache
- Un arbore binar are mai multe nivele (~8x), rezultând în mai multe umpleri de linii cache
- Un skip-list este aproximativ echivalent cu un arbore de grad 4, rezultând în mai multe umpleri de linii cache
- un arbore digital pe bază de întindere (a cărei variantă este Judy) nu necesită niciodată reechilibrări la creşterea arborelui
- o porţiune a cheii este utilizat pentru subdivizarea unei întinderi în sub-arbori. Doar restul cheii este trebuie să existe în sub-arbori, rezultând în compresia cheilor.

2.1.1 JudySL

JudySL este un vector asociativ, implementat folosing JudyL, în felul următor: se împarte un şir (terminat prin null), într-o secvenţă de cuvinte de 32/64 biţi lungime, şi se construieşte un arbore de vectori Judy, cu acele cuvinte ca şi indexi, reprezentând un prefix unic pentru fiecare şir. Fiecare nod terminal este un pointer la sufixul unic al şirului (un şir se poate termina şi fără nod terminal).

3 Contribuţii proprii la implementare de algoritmi de compresie

3.1 Coduri lungime-distanţă

În specificaţia *DEFLATE* se folosesc 28 de coduri pentru coduri de lungime, dar acesta se poate extinde în caz de nevoie.

Se observă următoarele la tabelul (1):

- primii 8 coduri au 0 extra biţi
- pentru următoarele 4 coduri avem *1* bit extra
- pentru următoarele 4 coduri avem *2* biţi extra
- ...
- pentru următoarele 4 coduri avem *i* biţi extra

Acesta conduce la următorul algoritm de generare a tabelului:

- primele 8 coduri, au 0 biţi extra
- $extra\_bits_i = \lfloor (i-8)/4 \rfloor$
- lungimile corespunzătoare unui cod $[start_i, end_i]$:
  - $start_i = end_{i-1} + 1$
  - $end_i = start + 2^{extra\_bits_i} - 1$

Acest algoritm l-am implementat în mk_codes.c ((A.1)).

3.2 Implementarea algoritmului LZ77 folosind arbori Judy

Pentru comprimarea datelor folosind coduri distanţă-lungime, elementul central este găsirea unei potriviri, de preferat a unei potriviri cât mai lungi.

Structura de date numită JudySL (secţiunea (2.1.1)) este foarte asemănătoare cu ce avem nevoie: asociază unui şir o valoare.

În cazul nostru valoarea este poziţia/distanţa, iar şirul sunt chiar datele din "sliding windowul" algoritmului LZ77. Singura problemă este că putem avea şi caracterul nul (0) în date, deci nu putem folosi JudySL ca atare.

Pe baza descrierii algoritmului JudySL din secţiunea (2.1.1) am realizat o structură de date, implementând următoarii operatori:

- judy_insert_bytearray(judyarray, data, length, position)
- judy_remove_bytearray(judyarray, data, length, position)
- judy_search_longestmatch(judyarray, data, length, &distance, &position)

Implementarea se află în lz_coder.c(A.2).

Bibliografie

[1] G.N.N Martin - "Range encoding: an algorithm for removing redundancy from a digitised message.", Video & Data Recording Conference, Southampton, 1979, http://www.compressconsult.com/rangecoder/rngcod.pdf.gz

[2] Alan Silverstein - "Judy IV Shop Manual", Hewlett-Packard, August 2002, http://judy.sourceforge.net/doc/shop_interm.pdf

[3] Doug Baskins - "A 10-minute description of how Judy arrays work and why they are so fast", http://judy.sourceforge.net/doc/10minutes.htm, July 2002

[4] * - http://judy.sourceforge.net/examples/judysl.pdf

[5] Peter Deutsch - "DEFLATE Compressed Data Format Specification version 1.3 (RFC 1951)", May 1996 - ftp://ftp.nic.it/rfc/rfc1951.pdf

[6] Mark Nelson - "Arithmetic Coding + Statistical Modeling = Data Compression", Dr. Dobb's Journal, February 1991, http://www.dogma.net/markn/articles/arith/part2.htm

[7] Moffat, Neal, and Witten - "Arithmetic Coding Revisited", ACM Transactions on Information Systems, July 1998, 16(3):256-294

[8] Matthew V. Mahoney - "Adaptive Weighing of Context Models for Lossless Data Compression", Florida Institute of Technology CS Dept, Technical Report CS-2005-16, https://www.cs.fit.edu/Projects/tech_reports/cs-2005-16.pdf

[9] Ziv J., Lempel A. - "A Universal Algorithm for Sequential Data Compression", IEEE Transactions on Information Theory, Vol. 23, No. 3, pp. 337-343. http://www.cs.duke.edu/courses/spring03/cps296.5/papers/ziv_lempel_1977_universal_algorithm.pdf

A Programul singularity-compress

Se găseşte şi la http://code.google.com/p/singularity-compress/ , http://singularity-compress.googlecode.com/svn/trunk/

```
/*
 * Singularity-compress: LZ77 encoder: length code tables
 * Copyright (C) 2006-2007      Torok Edwin (edwintorok@gmail.com)
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; version 2
 * of the License.

 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.

 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA                02110-1301, USA.
 */
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

static void make_length_code_tables(uint8_t extra_symbols,const ssize_t start)
{
    ssize_t code;
    unsigned short extra_bits = 0;
    unsigned short extra_bits_count = 0;
    ssize_t      i;

    printf("#include \"stdlib.h\"\n\n");
    printf("static const struct {\n");
    printf("\tssize_t start;\n");
    printf("\tuint8_t  extra_bits;\n");
    printf("} code_to_length[] = {\n");

    printf("\t");
    for(code = start; code < start + 8 && code < start + extra_symbols; code++)
        printf("{%lu, 0}, ",code);

    extra_bits = 1;
    printf("\n\t");
    for(i = 8; i < extra_symbols; i++ ) {
        printf("{%lu, %u}, ", code, extra_bits);

        code += (1 << extra_bits);
        extra_bits_count++;

        if ( extra_bits_count == 4) {
            printf("\n\t");
            extra_bits++;
            extra_bits_count=0;
        }
```

```
    }
    printf("{%ld, 0xff}\n",code);

    printf("};\n");
}


int main(int argc,char* argv[])
{
    printf("#ifndef _CODES_TABLE_H\n");
    printf("#define _CODES_TABLE_H\n\n");
    make_length_code_tables(28,3);
    printf("#endif\n");
    return 0;
}
```

A.2 lz_coder.c

```
/*
 * Singularity-compress: LZ77 encoder
 * Copyright (C) 2006-2007     Torok Edwin (edwintorok@gmail.com)
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; version 2
 * of the License.

 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.

 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA                02110-1301, USA.
 */
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef WIN32
#include <netinet/in.h>
#else

#define ntohl(x) bswap_32(x)
#endif

#include <stdarg.h>
#include "lz_coder.h"

/* ntohl uses bswap_32, as appropriate for current endianness,
 * also bswap_32 on libc/gcc uses asm instructions to swap bytes with 1 instruction */
#define CHAR4_TO_UINT32(data, i) ntohl((*(const uint32_t*)(&data[i])))
```

```
/* Use JLAP_INVALID to mark pointers to "not JudyL arrays", in this case pointers to Judy1 array
 * JLAP_INVALID is defined in Judy.h, and is currently 0x1. Since malloc returns a pointer at least
 * 4–byte
 * aligned, marking it with 0x1, allows us to differentiate Judy1, and JudyL pointers */

#define J1P_PUT( PJ1Array )    ( (Pvoid_t) ( (Word_t)(PJ1Array) | JLAP_INVALID) )
#define J1P_GET( PJ1Array )    ( (Pvoid_t) ( (Word_t)(PJ1Array) & ~JLAP_INVALID) )
#define IS_J1P(    PJ1Array )    ( (Word_t)(PJ1Array) & JLAP_INVALID )


/* offset –> WRAP(offset+buffer_len_half) : new buffer
 * WRAP(offset+buffer_len/2–1) –> offset–1: old buffer (search buffer)
 */

#define WRAP_BUFFER_INDEX(lz_buff, index) ((index) & (lz_buff–>buffer_len_mask))

/* error values */
#define EMEM –2



/* debug logging */
/*#define LOG_DEBUG 1*/


#if !defined(NDEBUG) && defined(LOG_DEBUG)
static void log_debug(const size_t line, const char* fmt,...)
{
    va_list ap;
    va_start(ap,fmt);
    fprintf(stderr,"DEBUG %s: %ld ",__FILE__,line);
    vfprintf(stderr,fmt,ap);
    va_end(ap);
}

#else
static inline void log_debug(const size_t line, const char* fmt,...) {}
#endif

/*  ********** */
int setup_lz_buffer(struct lz_buffer* lz_buffer,const size_t buffer_len_power)
{
    lz_buffer–>buffer_len_power = buffer_len_power;
    lz_buffer–>buffer_len       = 1 << buffer_len_power;
    lz_buffer–>buffer_len_mask  = lz_buffer–>buffer_len – 1;
    lz_buffer–>offset           = 0;
    lz_buffer–>jarray           = (Pvoid_t) NULL;
    lz_buffer–>buffer = calloc(lz_buffer–>buffer_len+4,1);
    if(!lz_buffer–>buffer) {
        log_debug(__LINE__,"Out of memory while trying to allocate %ld bytes",
          lz_buffer–>buffer_len);
        return EMEM;
    }
```

```
    return 0;
}


static void judy_free_tree(Pvoid_t jarray)
{
    Word_t Index = 0;

    if(!jarray)
        return;

    if(IS_J1P(jarray)) {
        int rc;
        Pvoid_t judy1_node = J1P_GET(jarray);

        log_debug(__LINE__,"Retrieved original Judy1 array pointer: %p from %p, \
freeing it.\n",judy1_node, jarray);

        J1FA(rc, judy1_node);
    }
    else {
        Word_t *PValue;

        log_debug(__LINE__,"Retrieving first entry in JudyL array: %p. ",jarray);

        JLF(PValue, jarray, Index);

        log_debug(__LINE__,"Retrieved first entry: index: %lx; value: %p -> %p\n\
", Index, PValue, *PValue);

        while(PValue != NULL) {
            log_debug(__LINE__,"At entry: index: %lx, value: %p -> %p\n", Index, PValue,
              *PValue);

            judy_free_tree((Pvoid_t)*PValue);
            JLN(PValue,jarray, Index);
        }
        {
            int rc;
            log_debug(__LINE__,"Freeing JudyL array: %p\n",jarray);
            JLFA(rc,jarray);
        }
    }
}

void cleanup_lz_buffer(struct lz_buffer* lz_buffer)
{
    if(lz_buffer->buffer) {
        free(lz_buffer->buffer);
        lz_buffer->buffer = NULL;
    }
    if(lz_buffer->jarray)
        judy_free_tree(lz_buffer->jarray);
}
```

```
static Pvoid_t create_judy_tree(const struct lz_buffer* lz_buff,const size_t offset, const ssize_t
 length,const size_t position)
{
    Pvoid_t    PJLArray = (Pvoid_t) NULL;
    Pvoid_t* const first_node = &PJLArray;
    Pvoid_t* node = first_node;
    ssize_t i;

    for(i=0; i < length-4 ; i += 4) {
        Pvoid_t* next_node;
        const uint32_t val = CHAR4_TO_UINT32(lz_buff->buffer, WRAP_BUFFER_INDEX(lz_buff,
         offset+i) );

        log_debug(__LINE__,"create_judy_tree: Inserting %x into JudyL array: %p \
-> %p\n",val, node, *node);

        JLI(next_node, *node, val );

        log_debug(__LINE__,"create_judy_tree: Inserted into %p->%p, value: %p \
-> %p\n", node, *node, next_node, *next_node);

        if (next_node == PJERR)
            return PJERR;
              node = (Pvoid_t*) next_node;
    }
    {
        int rc;

        log_debug(__LINE__,"create_judy_tree: Inserting into judy1 array \
%p->%p, value:%lx\n",node,*node,position);

        J1S(rc, *node, position );

        log_debug(__LINE__,"create_judy_tree: Inserted into %p->%p\n",node,*node);

        if(rc == JERR)
            return PJERR;

        *node = J1P_PUT(*node);
        if(i==0)
            return *node;
    }
    return *first_node;
}


static int judy_remove_bytearray(const struct lz_buffer* lz_buff,const size_t offset,const size_t
 length,Pvoid_t* node,size_t position)
{

    memcpy(&lz_buff->buffer[lz_buff->buffer_len],&lz_buff->buffer[0],4);
```

```
    if( length>0 && !IS_J1P(*node) ) {
        int rc;
        Pvoid_t* next;
        const uint32_t val = CHAR4_TO_UINT32(lz_buff->buffer, WRAP_BUFFER_INDEX(lz_buff,
         offset) );

        log_debug(__LINE__,"judy_remove_bytearray: Querying %x, in JudyL array \
%p->%p.   ",val,node,*node);

        JLG(next,*node, val);

        log_debug(__LINE__,"Query result: %p->%p\n", next, next==NULL ? NULL : *next);

        if(next == NULL) {
            log_debug(__LINE__,"judy_remove_bytearray: not found:%x!\n",val);
            return -1;
        }
        rc = judy_remove_bytearray(lz_buff, offset + 4, length-4, next, position);
        if(!*next) {
            JLD(rc, *node,val);
        }
        return rc;
    } else if( IS_J1P(*node)) {
        int rc;
        Pvoid_t judy1_node    = J1P_GET(*node);

        log_debug(__LINE__,"Retrieved Judy1 pointer %p from %p->%p.Removing \
value: %lx\n",judy1_node,node,*node, position);

        J1U(rc,judy1_node, position );
        if(!rc) {
            log_debug(__LINE__,"judy_remove_bytearay: not Found pos:%ld!\n",position);
        }

        log_debug(__LINE__,"Storing Judy1 array into %p->%p\n",node,*node);

        if(judy1_node) {
            *node = J1P_PUT(judy1_node);
            if(rc == JERR)
                return JERR;
        } else {
            *node = NULL;
        }
    }
    return 0;
}


int lz_remove(struct lz_buffer* lz_buff,const size_t offset)
{
    return judy_remove_bytearray(lz_buff,offset,lz_buff->buffer_len/2,&lz_buff->jarray,
     WRAP_BUFFER_INDEX(lz_buff, offset));
}
```

```
/* length must be multiple of 4 */
static int judy_insert_bytearray(const struct lz_buffer* lz_buff,const size_t offset, const size_t length,
 Pvoid_t* node,size_t position)
{
    size_t i;


    memcpy(&lz_buff->buffer[lz_buff->buffer_len],&lz_buff->buffer[0],4);

    for(i=0;i < length && !IS_J1P(*node); i += 4) {
        Pvoid_t* next;
        const uint32_t val = CHAR4_TO_UINT32(lz_buff->buffer, WRAP_BUFFER_INDEX(lz_buff,
         offset + i) );

        log_debug(__LINE__,"judy_insert_bytearray: Querying %x, in JudyL array \
%p->%p.  ",val,node,*node);

        JLG(next,*node, val);

        log_debug(__LINE__,"Query result: %p->%p\n", next, next==NULL ? NULL : *next);

        if(next == NULL) {
            log_debug(__LINE__,"judy_insert_bytearray: Inserting %x, into \
%p->%p.\n",val,node,*node);

            JLI(next, *node,val);

            log_debug(__LINE__,"judy_insert_bytearray: Inserted into %p->%p, \
value: %p->%p\n",node,*node,next,*next);

            *next = create_judy_tree(lz_buff, offset+i+4, length-i-4, position);

            log_debug(__LINE__,"judy_insert_bytearray: Stored into \
:%p->%p;value: %p->%p\n",node,*node,next,*next);

            if(*next == PJERR)
                return JERR;
            else
                return 0;
        }
        node = next;
    }

    if(IS_J1P(*node)) {
        int rc;
        Pvoid_t judy1_node    = J1P_GET(*node);

        log_debug(__LINE__,"Retrieved Judy1 pointer %p from %p->%p. Setting \
value: %lx\n",judy1_node,node,*node, position);

        J1S(rc,judy1_node, position );

        log_debug(__LINE__,"Storing Judy1 array into %p->%p\n",node,*node);
```

```
        *node = J1P_PUT(judy1_node);
        if(rc == JERR)
            return JERR;
    }
    return 0;
}

int lzbuff_insert(struct lz_buffer* lz_buff, const char c)
{
    const int rc = judy_insert_bytearray(lz_buff, lz_buff->offset, lz_buff->buffer_len/2, &lz_buff
     ->jarray, lz_buff->offset);
    lz_buff->offset = WRAP_BUFFER_INDEX(lz_buff, lz_buff->offset + 1 );
    /*lz_buff->buffer[ lz_buff->offset ] = c;*/
    return rc;
}

static int get_longest_match(const struct lz_buffer * lz_buff, const uint32_t prev,const uint32_t next,
 size_t pos,ssize_t* distance, ssize_t* length)
{
    size_t i;
    const size_t buffer_half_len = lz_buff->buffer_len/2;
    const ssize_t data_search_idx = lz_buff->offset - buffer_half_len/2 - 1;
    ssize_t data_buff_idx        = prev;

    size_t match_len = pos;

    for(i = pos; i < buffer_half_len; i++) {
        if(lz_buff->buffer[WRAP_BUFFER_INDEX(lz_buff, data_search_idx + i)] != lz_buff
         ->buffer[WRAP_BUFFER_INDEX(lz_buff, data_buff_idx + i)]) {
            match_len = i - 1;
            break;
        }
    }

    if(i == buffer_half_len) {
        /* entire buffer matches */
        *length = buffer_half_len;
        *distance = WRAP_BUFFER_INDEX(lz_buff, lz_buff->offset - prev);
        return 0;
    }

    data_buff_idx = next;

    for(i = pos; i < buffer_half_len; i++) {
        if(lz_buff->buffer[WRAP_BUFFER_INDEX(lz_buff, data_search_idx + i)] != lz_buff
         ->buffer[WRAP_BUFFER_INDEX(lz_buff, data_buff_idx + i)]) {
            if(i - 1 < match_len) {
                /* data_buff_idx = prev has a longer match */
                *distance = WRAP_BUFFER_INDEX(lz_buff, lz_buff->offset - prev);
            }
            else {
                match_len = i-1;
                *distance = WRAP_BUFFER_INDEX(lz_buff, lz_buff->offset - next);
```

```
            }
            *length = match_len;
            return 0;
        }
    }

    /* entire buffer matches */
    *length = buffer_half_len;
    *distance = WRAP_BUFFER_INDEX(lz_buff, lz_buff->offset - prev);
    return 0;
}


/* return number of bytes that match,
 * we always assume big endian, since
 * 0x30313233, comes from the string "0123" */
static inline uint8_t compare_bytes(const uint32_t a,const uint32_t b)
{
    const uint32_t c = a ^ b;/*will be 0 where there is a match*/
    if(c&0xff000000) /* if there are any 1's there, we got a mismatch in the very first byte */
        return 0;
    if(c&0x00ff0000)
        return 1;
    if(c&0x0000ff00)
        return 2;
    if(c&0x000000ff)
        return 3;
    return 4;
}


static inline const Pvoid_t* get_best_match(const uint32_t orig_val, const Word_t prev_val, const
 Word_t next_val, const Pvoid_t* prev, const Pvoid_t* next,uint8_t* match)
{
    /* determine how many bytes we have matched */
    const uint8_t match_next = compare_bytes(next_val, orig_val);
    const uint8_t match_prev = compare_bytes(prev_val, orig_val);
    if( match_prev < match_next) {
        *match = match_next;
        return next;
    }

    *match = match_prev;
    return prev;
}


/* gets index that is closest */

#define J1_CLOSEST(rc, J1Array, search_index,       prev_index, next_index) \
{\
    (next_index) = (search_index);\
    J1F((rc), (J1Array), (next_index));\
    if(!(rc)) {\
```

```
        (prev_index) = (search_index);\
        J1P((rc), (J1Array), (prev_index));\
        if((rc)) {\
            (next_index) = (prev_index);\
            J1N((rc), (J1Array), (next_index));\
        }\
    }\
    else {\
        if((next_index) != (search_index)) {\
            (prev_index) = (next_index);\
            J1P((rc), (J1Array), (prev_index));\
        }\
    }\
}

#define J1_NEIGHBOUR(rc, J1Array, search_index, neighbour_index) \
{\
    (neighbour_index) = (search_index);\
    J1F((rc), (J1Array), (neighbour_index));\
    if(!(rc)) {\
        (neighbour_index) = (search_index);\
        J1P((rc), (J1Array), (neighbour_index));\
    }\
}

static int get_closest(const struct lz_buffer* lz_buff,const Pvoid_t* node,ssize_t* distance)
{
    /* now walk the array of JudyL arrays, till we reach judy1 pointers, and then select the
     * position stored there that is closest to current offset*/
    while(node && !IS_J1P(*node)) {
        Pvoid_t *next;
        Word_t val = 0;
        JLF(next,*node,val);
        node = next;
    }

    if(node == NULL) {
        fprintf(stderr,"Warning: encountered unexpected empty JudyL array");
        *distance = 0;
        return -2;
    }
    else {
        int rc;
        Pvoid_t judy1_node = J1P_GET(*node);
        Word_t val ;

        J1_NEIGHBOUR(rc, judy1_node, lz_buff->offset, val);

        if(!rc) {
            fprintf(stderr,"Emtpy judy1 array?\n");
            *distance = 0;
            return -1;
        }
```

```
        *distance = WRAP_BUFFER_INDEX(lz_buff, lz_buff->offset - val);
        return 0;
    }
}


int lzbuff_search_longest_match(const struct lz_buffer* lz_buff,const size_t offset,const size_t
 data_len, ssize_t* distance, ssize_t* length)
{
    size_t i;
    /* start searching from root array */
    const Pvoid_t* node = &lz_buff->jarray;

    memcpy(&lz_buff->buffer[lz_buff->buffer_len],&lz_buff->buffer[0],4);
    /* when we reach end-of-buffer during a search, we would need to wrap the search,
     * but we can only wrap on multiples of 4, so to prevent accessing uninitialized memory,
     * we copy first 4 bytes, to end of buffer */

    for(i=0;i < data_len && !IS_J1P(*node); i+=4) {
        const Pvoid_t* next;
        const Pvoid_t* prev;

        const uint32_t orig_val = CHAR4_TO_UINT32(lz_buff->buffer,
         WRAP_BUFFER_INDEX(lz_buff, offset + i) );
        /* indexes in the array are groups of 4 characters converted into uint32 */
        Word_t prev_val;
        Word_t next_val = orig_val;

        /* search the first index >= than the value we search for,
         * or if it doesn't exist, then the index lower than it
         * next is the value contained in the JudyL array at index val
         * It is really a pointer to another JudyL array (if IS_J1P(*next) == 0) ;
         * Or it is a pointer to a Judy1 array if IS_J1P(*next) == 1
         */

        JLF( next, *node, next_val);
        if(!next) {
            prev_val = orig_val;
            JLP(prev, *node, prev_val);
            if(prev) {
                /* previous found, search for next neighbour now */
                next_val = prev_val;
                JLN(next, *node, next_val);
                if(!next) {
                    /* array has only 1 element */
                    next = prev;
                    next_val = prev_val;
                }
            }
            else {
                log_debug(__LINE__,"Empty JudyL array?\n");
                /* no value found in JudyL array, buffer is empty */
                *length = -1;
                *distance = 0;
```

```
            return -1;
        }
    }
    else {
        if(next_val != orig_val) {
            prev_val = next_val;
            JLP(prev, *node, prev_val);
            if(!prev) {
                /* array has only 1 element */
                prev = next;
                prev_val = next_val;
            }
        }
    }

    if(next_val != orig_val) {
        uint8_t match;
        /* we encountered a mismatch */
        node = get_best_match(orig_val, prev_val, next_val, prev, next ,&match);
        *length = match+i;
        return get_closest(lz_buff, node, distance);
    }

    node = next;
}

if(IS_J1P(*node)) {
    int rc;
    Pvoid_t judy1_node = J1P_GET(*node);
    const uint32_t orig_val = CHAR4_TO_UINT32(lz_buff->buffer,
     WRAP_BUFFER_INDEX(lz_buff, offset + i) );
    Word_t val = orig_val;
    Word_t val_prev, val_next;

    log_debug(__LINE__,"Retrieved original Judy1 array pointer: %p from \
%p->%p\n",judy1_node, node,*node);

    J1_CLOSEST(rc, judy1_node, val, val_prev, val_next);

    if(val_next != orig_val) {
        return get_longest_match(lz_buff,val_prev, val_next,i, distance, length);
    }
    else {
        *distance = -data_len;
        *length   = data_len;
        return 0;
    }
}
else {
    return -1;
    /* assertion failure */
}
}
```

```
static void judy_show_tree(Pvoid_t jarray,int level)
{
    char* spaces = malloc(level+1);
    Word_t Index = 0;

    memset(spaces,0x20,level);
    spaces[level] = 0;

    if(IS_J1P(jarray)) {
        int rc;
        Pvoid_t judy1_node = J1P_GET(jarray);
        J1F(rc,judy1_node,Index);
        while(rc) {
            fprintf(stderr,":%s%lx\n",spaces,Index);
            J1N(rc,judy1_node,Index);
        }
    }
    else {
        Word_t * PValue;
        log_debug(__LINE__,"querying first:%p\n",jarray);
        JLF(PValue, jarray, Index);
        while(PValue != NULL) {
            fprintf(stderr,"%s%lx\n",spaces,Index);
            judy_show_tree((Pvoid_t)*PValue, level+1);
            JLN(PValue, jarray, Index);
        }
    }
    free(spaces);
}

void show_lz_buff(const struct lz_buffer* lz_buff)
{
    size_t i;
    for(i=0;i < lz_buff->buffer_len;i++) {
        if(i == lz_buff->offset)
            fprintf(stderr,"|");
        fprintf(stderr,"%02x ",lz_buff->buffer[i]);
    }
    fprintf(stderr,"\n");
}

void show_match(const struct lz_buffer* lz_buff,ssize_t distance,ssize_t length)
{
    size_t start = WRAP_BUFFER_INDEX(lz_buff, lz_buff->offset - distance);

    ssize_t match = -1;
    size_t i;

    for(i=0;i < lz_buff->buffer_len;i++) {
        if(i == start) {
            fprintf(stderr,"<");
            match = 0;
        }
```

```
        if(match >= 0)
            match++;
        fprintf(stderr,"%02x ",lz_buff->buffer[i]);
        if(match == length)
            printf(">");
    }
    fprintf(stderr,"\n");
}

/*
int main(int argc,char* argv[])
{
    const unsigned char test[] = "0123456789012345";
    const size_t len = sizeof(test)-1;
    size_t i;

    struct lz_buffer        lz_buff;
    FILE* out = fopen("/tmp/testout","w");

    setup_lz_buffer(&lz_buff,5);


     this will be replaced by a read()
    for(i=0;i < len;i++) {
        lz_buff.buffer[lz_buff.offset + i] = test[i];
    }


    for(i=0; i < len;) {
        ssize_t distance;
        ssize_t     length;

        show_lz_buff(&lz_buff);

        lzbuff_search_longest_match(&lz_buff, &lz_buff.buffer[lz_buff.offset], lz_buff.buffer_len/2,
        &distance, &length);

        show_match(&lz_buff,distance,length);

        lzbuff_insert(&lz_buff, lz_buff.buffer[lz_buff.offset + i]);

        show_lz_buff(&lz_buff);
        judy_show_tree(lz_buff.jarray,0);
        printf("match:%ld,%ld\n",distance,length);
        if(length < 3) {
            length = 1;
        }
        else {
        }
        i += length;
    }
    cleanup_lz_buffer(&lz_buff);

    return 0;
```

```
}
 */
/*
int main(int argc,char* argv[])
{
    Pvoid_t jarray = (Pvoid_t) NULL;
    const size_t len = 16*2;
    const size_t maxIdx = 8;
    unsigned char* test = malloc(len+1);

    const unsigned char test2[] = "0123456789012346";
    const size_t len2 = sizeof(test)-1;
    size_t i;
    ssize_t distance=0;
    size_t length=0;

    srand(2);
    for(i=0;i<len/2;i++) {
        test[i] = test2[i%len2];
    }
    test[len]=0;
    for(i=0;i<len/2;i++) {
        judy_insert_bytearray(test+i,maxIdx,&jarray,i);
    }
    judy_insert_bytearray(test,maxIdx,&jarray,0);

    judy_show_tree(jarray,0);
    struct lz_buffer buff;
    lzbuff_search_longest_match(&buff,test2+10,len2-10,&distance,&length);
    printf("%ld:%ld\n",distance,length);
    judy_free_tree(jarray);
    jarray = NULL;
    return 0;
}
 */
```

A.3 lz_coder.h

```
/*
 * Singularity-compress: LZ77 encoder
 * Copyright (C) 2006-2007      Torok Edwin (edwintorok@gmail.com)
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; version 2
 * of the License.

 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.

 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
```

```
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.
 */
#ifndef _LZ_CODER_H
#define _LZ_CODER_H

#include <Judy.h>

struct lz_buffer {
    unsigned char*      buffer;
    size_t      buffer_len;
    size_t      buffer_len_mask;
    size_t      buffer_len_power;/* buffer_len = 2^buffer_len_base2 */
    ssize_t      offset;
    Pvoid_t  jarray;
};
int setup_lz_buffer(struct lz_buffer* lz_buffer,const size_t buffer_len_power);
void cleanup_lz_buffer(struct lz_buffer* lz_buffer);

int lzbuff_insert(struct lz_buffer* lz_buff, const char c);
int lzbuff_search_longest_match(const struct lz_buffer* lz_buff,const size_t offset,const size_t
 data_len, ssize_t* distance, ssize_t* length);

int lz_remove(struct lz_buffer* lz_buff,const size_t offset);
/* offset -> WRAP(offset+buffer_len_half) : new buffer
 * WRAP(offset+buffer_len/2-1) -> offset-1: old buffer (search buffer)
 */
#define WRAP_BUFFER_INDEX(lz_buff, index) ((index) & (lz_buff->buffer_len_mask))

/* debugging functions - to be removed in a release */
void show_lz_buff(const struct lz_buffer* lz_buff);
void show_match(const struct lz_buffer* lz_buff,ssize_t distance,ssize_t length);
#endif
```

A.4 range_encoder.h

```
/*
 * Singularity-compress: rangecoder encoder
 * Copyright (C) 2006-2007      Torok Edwin (edwintorok@gmail.com)
 *
 * Based on Michael Schindler's rangecoder, which is:
 * (c) Michael Schindler 1997, 1998, 1999, 2000
 * http://www.compressconsult.com/
 *
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; version 2
 * of the License.

 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
```

```
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110–1301, USA.
 */

#ifndef _RANGE_ENCODER_H
#define _RANGE_ENCODER_H
#include "../common/rangecod.h"

/* rc is the range coder to be used                                */
/* c is written as first byte in the datastream                    */
/* one could do without c, but then you have an additional if      */
/* per outputbyte.                                                 */
static void start_encoding( rangecoder *rc, char c, int initlength )
{
    rc->low = 0;                    /* Full code range */
    rc->range = Top_value;
    rc->buffer = c;
    rc->help = 0;                   /* No bytes to follow */
    rc->bytecount = initlength;
}

static inline void enc_normalize( rangecoder *rc )
{
    while(rc->range <= Bottom_value) {
        /* do we need renormalisation?      */
            if (rc->low < (code_value)0xff << SHIFT_BITS) {
            /* no carry possible --> output */
            outbyte(rc, rc->buffer);
            for(; rc->help; rc->help--)
                outbyte(rc, 0xff);
            rc->buffer = (unsigned char)(rc->low >> SHIFT_BITS);
        } else if (rc->low & Top_value) {
            /* carry now, no future carry */
            outbyte(rc,rc->buffer+1);
            for(; rc->help; rc->help--)
                outbyte(rc,0);
            rc->buffer = (unsigned char)(rc->low >> SHIFT_BITS);
        } else    {
            /* passes on a potential carry */
#ifndef DO_CHECKS
            rc->help++;
#else
            if (rc->bytestofollow++ == 0xffffffffL) {
                fprintf(stderr,"Too many bytes outstanding - File too large\n");
                exit(1);
            }
#endif
        }

        rc->range <<= 8;
        rc->low = (rc->low<<8) & (Top_value-1);
        rc->bytecount++;
    }
```

```
}


/* Encode a symbol using frequencies                                    */
/* rc is the range coder to be used                                     */
/* sy_f is the interval length (frequency of the symbol)                */
/* lt_f is the lower end (frequency sum of < symbols)                   */
/* tot_f is the total interval length (total frequency sum)             */
/* or (faster): tot_f = (code_value)1<<shift                                    */
static void encode_freq( rangecoder *rc, freq sy_f, freq lt_f, freq tot_f )
{
    code_value r, tmp;
    enc_normalize( rc );
    r = rc->range / tot_f;
    tmp = r * lt_f;
    rc->low += tmp;
    if (lt_f+sy_f < tot_f)
        rc->range = r * sy_f;
    else
        rc->range -= tmp;
#ifdef DO_CHECKS
    if(!rc->range)
        fprintf(stderr,"ooops, zero range\n");
#endif
}

static void encode_shift( rangecoder *rc, freq sy_f, freq lt_f, freq shift )
{
    code_value r, tmp;
    enc_normalize( rc );
    r = rc->range >> shift;
    tmp = r * lt_f;
    rc->low += tmp;
    if ((lt_f+sy_f) >> shift)
        rc->range -= tmp;
    else
        rc->range = r * sy_f;
#ifdef DO_CHECKS
    if(!rc->range)
        fprintf(stderr,"Oops, zero range\n");
#endif
}


static uint32_t done_encoding( rangecoder *rc )
{
    size_t tmp;
    enc_normalize(rc);          /* now we have a normalized state */
    rc->bytecount += 5;
    if ((rc->low & (Bottom_value-1)) < ((rc->bytecount&0xffffffL)>>1))
        tmp = rc->low >> SHIFT_BITS;
    else
        tmp = (rc->low >> SHIFT_BITS) + 1;
    if (tmp > 0xff) /* we have a carry */
```

```
    {
        outbyte(rc,rc−>buffer+1);
        for(; rc−>help; rc−>help−−)
            outbyte(rc,0);
    } else    /∗ no carry ∗/
    {
        outbyte(rc,rc−>buffer);
        for(; rc−>help; rc−>help−−)
            outbyte(rc, 0xff);
    }
    outbyte(rc, tmp & 0xff);
        outbyte(rc, (rc−>bytecount≫16) & 0xff);
        outbyte(rc, (rc−>bytecount≫8) & 0xff);
        outbyte(rc, rc−>bytecount & 0xff);
    return rc−>bytecount;
}

#define encode_short(ac,s) encode_shift(ac,(freq)1,(freq)(s),(freq)16)

#endif
```

A.5 range_decod.h

```
/∗
 ∗ Singularity−compress: rangecoder decoder
 ∗ Copyright (C) 2006−2007    Torok Edwin (edwintorok@gmail.com)
 ∗
 ∗ Based on Michael Schindler's rangecoder, which is:
 ∗ (c) Michael Schindler 1997, 1998, 1999, 2000
 ∗ http://www.compressconsult.com/
 ∗
 ∗
 ∗ This program is free software; you can redistribute it and/or
 ∗ modify it under the terms of the GNU General Public License
 ∗ as published by the Free Software Foundation; version 2
 ∗ of the License.

 ∗ This program is distributed in the hope that it will be useful,
 ∗ but WITHOUT ANY WARRANTY; without even the implied warranty of
 ∗ MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 ∗ GNU General Public License for more details.

 ∗ You should have received a copy of the GNU General Public License
 ∗ along with this program; if not, write to the Free Software
 ∗ Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA    02110−1301, USA.
 ∗/

#ifndef _RANGE_DECOD_H
#define _RANGE_DECOD_H

#ifndef NDEBUG
#define DO_CHEKCS
#endif
```

```
#include "../common/rangecod.h"

/* Start the decoder                                   */
/* rc is the range coder to be used                    */
/* returns the char from start_encoding or EOF         */
static inline int start_decoding( rangecoder *rc )
{
    int c = inbyte(rc);
    if (c==EOF)
        return EOF;
    rc->buffer = inbyte(rc);
    rc->low = rc->buffer >> (8-EXTRA_BITS);
    rc->range = (code_value)1 << EXTRA_BITS;
    return c;
}

static inline void dec_normalize( rangecoder *rc )
{
    while (rc->range <= Bottom_value)     {
        rc->low = (rc->low<<8) | ((rc->buffer<<EXTRA_BITS)&0xff);
        rc->buffer = inbyte(rc);
        rc->low |= rc->buffer >> (8-EXTRA_BITS);
        rc->range <<= 8;
    }
}


/* Calculate culmulative frequency for next symbol. Does NO update!*/
/* rc is the range coder to be used                    */
/* tot_f is the total frequency                        */
/* or: totf is (code_value)1<<shift                                */
/* returns the culmulative frequency                   */
static inline freq decode_culfreq( rangecoder *rc, freq tot_f )
{
    freq tmp;
    dec_normalize(rc);
    rc->help = rc->range/tot_f;
    tmp = rc->low/rc->help;
    return (tmp>=tot_f ? tot_f-1 : tmp);
}



static inline freq decode_culshift( rangecoder *rc, freq shift )
{
    freq tmp;
    dec_normalize(rc);
    rc->help = rc->range>>shift;
        tmp = rc->low/rc->help;
    return (tmp>>shift ? ((code_value)1<<shift)-1 : tmp);
}


/* Update decoding state                               */
```

```
/* rc is the range coder to be used                                    */
/* sy_f is the interval length (frequency of the symbol)               */
/* lt_f is the lower end (frequency sum of < symbols)                  */
/* tot_f is the total interval length (total frequency sum)            */
static inline void decode_update( rangecoder *rc, freq sy_f, freq lt_f, freq tot_f)
{
    code_value tmp;
    tmp = rc->help * lt_f;
    rc->low -= tmp;
    if (lt_f + sy_f < tot_f)
        rc->range = rc->help * sy_f;
    else
        rc->range -= tmp;
}

#define decode_update_shift(rc,f1,f2,f3) decode_update((rc),(f1),(f2),(freq)1<<(f3));

/* Decode a byte/short without modelling                               */
/* rc is the range coder to be used                                    */
static inline unsigned char decode_byte(rangecoder *rc)
{
    unsigned char tmp = decode_culshift(rc,8);
    decode_update( rc,1,tmp,(freq)1<<8);
    return tmp;
}

static inline unsigned short decode_short(rangecoder *rc)
{
    unsigned short tmp = decode_culshift(rc,16);
    decode_update( rc,1,tmp,(freq)1<<16);
    return tmp;
}

/* Finish decoding                                                     */
/* rc is the range coder to be used                                    */
static inline void done_decoding( rangecoder *rc )
{
    dec_normalize(rc);          /* normalize to use up all bytes */
}

#endif
```

A.6 codes.h

```
/*
 * Singularity-compress: LZ77 encoder: length code tables
 * Copyright (C) 2006-2007      Torok Edwin (edwintorok@gmail.com)
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; version 2
 * of the License.

 * This program is distributed in the hope that it will be useful,
```

```
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.

 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110–1301, USA.
 */
#ifndef _CODES_H
#define _CODES_H

#include "stdlib.h"

static inline uint16_t length_to_code(ssize_t length,uint8_t* extra_bits,size_t* extra_data)
{
    size_t low = 0;
    size_t hi  = code_to_length_size–1;

    while(low < hi) {
        const size_t middle = (hi+low)/2;
        if(code_to_length[middle].start > length) {
            hi = middle–1;
        }
        else if(code_to_length[middle].start < length) {
            low = middle+1;
        }
        else {
            *extra_data = length – code_to_length[middle].start;
            *extra_bits = code_to_length[middle].extra_bits;

            return middle;
        }
    }

    if(code_to_length[low].start > length)
        low––;

    *extra_data = length – code_to_length[low].start;
    *extra_bits = code_to_length[low].extra_bits;

    return low;
}

#endif
```

A.7 code_tables.h

```
#ifndef _CODES_TABLE_H
#define _CODES_TABLE_H

static const struct {
    ssize_t start;
    uint8_t  extra_bits;
} code_to_length[] = {
```

```
    {3, 0}, {4, 0}, {5, 0}, {6, 0}, {7, 0}, {8, 0}, {9, 0}, {10, 0},
    {11, 1}, {13, 1}, {15, 1}, {17, 1},
    {19, 2}, {23, 2}, {27, 2}, {31, 2},
    {35, 3}, {43, 3}, {51, 3}, {59, 3},
    {67, 4}, {83, 4}, {99, 4}, {115, 4},
    {131, 5}, {163, 5}, {195, 5}, {227, 5},
    {259, 0xff}
};

static const size_t code_to_length_size = sizeof(code_to_length)/sizeof(code_to_length[0]);
#endif
```

A.8 model.h

```
/*
 * Singularity–compress: statistical model
 * Copyright (C) 2006–2007      Torok Edwin (edwintorok@gmail.com)
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; version 2
 * of the License.

 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.

 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA                02110–1301, USA.
 */
#ifndef _MODEL_H
#define _MODEL_H

/* keep the blocksize below 1<<16 or you'll see overflows */
#define BLOCKSIZE_POWER 10
#define BLOCKSIZE (1<<BLOCKSIZE_POWER)


#define EXTRA_SYMBOLS 28
#define SYMBOLS (256 + EXTRA_SYMBOLS)

struct ari_model {
    size_t* counts;
};

static void model_setup(struct ari_model* model)
{
    size_t i;
    model->counts = malloc(sizeof(model->counts[0])*(SYMBOLS+1));
    for(i=0;i < SYMBOLS+1;i++) {
            model->counts[i]=i;
    }
```

```
}

static void model_done(struct ari_model* model)
{
    free(model->counts);
}

static void model_get_freq(const struct ari_model* model,const uint16_t symbol,freq* cur_freq,freq*
 cum_freq,freq* total_freq)
{
    *cur_freq = model->counts[symbol+1] - model->counts[symbol];
    *cum_freq = model->counts[symbol];
    *total_freq = model->counts[SYMBOLS];
}

static void model_update_freq(struct ari_model* model,const uint16_t symbol)
{
    size_t i;
    /*fprintf(stderr,"Updating:%d : %ld\n",symbol,model->counts[symbol]);*/
    for(i=symbol;i <= SYMBOLS; i++) {
        model->counts[i]++;
    }
}

static freq model_get_symbol(const struct ari_model* model,const freq cf)
{
    const size_t* counts = model->counts;
    size_t sym_lo = 0;
    size_t sym_hi = SYMBOLS;

    do{
        size_t middle = (sym_lo+sym_hi+1)/2;
        /* we need +1 there, because we want last symbol that has freq <= cf */
        if(counts[middle] > cf) {
            sym_hi = middle - 1;
        }
        else if(counts[middle] < cf) {
            sym_lo = middle + 1;
        }
        else {
            sym_lo = middle;
        }

    } while(sym_lo < sym_hi);
    if(counts[sym_lo] > cf)
        sym_lo--;
    return sym_lo;
}
#endif
```

A.9 rangecod.h

```
/*
    rangecod.h          headerfile for range encoding
```

```
   (c) Michael Schindler
   1997, 1998, 1999, 2000
   http://www.compressconsult.com/
   michael@compressconsult.com

   This program is free software; you can redistribute it and/or
   modify it under the terms of the GNU General Public License
   as published by the Free Software Foundation; version 2
   of the License.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110–1301, USA.
*/

#ifndef _RANGECOD_H
#define _RANGECOD_H

typedef uint32_t code_value;            /* Type of an rangecode value, must accomodate 32 bits */
/* it is highly recommended that the total frequency count is less  */
/* than 1 << 19 to minimize rounding effects.                       */
/* the total frequency count MUST be less than 1<<23                */

typedef uint32_t freq;


typedef struct {
    uint32_t low,           /* low end of interval */
        range,              /* length of interval */
        help;               /* bytes_to_follow resp. intermediate value */
    unsigned char buffer;/* buffer for input/output */
    /* the following is used only when encoding */
    uint32_t bytecount;         /* counter for outputed bytes       */
    FILE*    in;
    /* insert fields you need for input/output below this line! */
} rangecoder;

#define CODE_BITS 32
#define Top_value ((code_value)1 << (CODE_BITS-1))

#define outbyte(cod,x)  putchar(x)
#define inbyte(cod)     getc(cod->in)

#define SHIFT_BITS (CODE_BITS - 9)
#define EXTRA_BITS ((CODE_BITS-2) % 8 + 1)
#define Bottom_value (Top_value >> 8)

#endif
```

A.10 simple_c.c

```
/*
 * Singularity–compress: arithmetic encoder
 *
 * Based on Michael Schindler's rangecoder, which is:
 * (c) Michael Schindler 1999
 * http://www.compressconsult.com/
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; version 2
 * of the License.

 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.

 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110–1301, USA.
 */

#include <stdio.h>
#include <stdlib.h>

#ifndef unix
#include <fcntl.h>
#endif

#include <ctype.h>
#include <stdint.h>
#include <string.h>

#include "range_encoder.h"

#include "lz_coder.h"
#include "../common/code_tables.h"
#include "codes.h"
#include "../common/model.h"

/*
 * Symbols:
 * 0–255: literal
 * 256–288: match length: 3–
 *
 */


struct lz_extra_data {
    uint8_t     extra_bits;
    size_t      extra_data;
    uint16_t    distance;
```

```
};


static void usage(void)
{
    fprintf(stderr,"simple_c [inputfile [outputfile]]\n");
    exit(1);
}

static int errcnt=0;
static int check_lz(const struct lz_buffer* lz_buff,ssize_t length,ssize_t distance)
{
    ssize_t i;
    for(i=0;i<length;i++)
        if(lz_buff->buffer[ WRAP_BUFFER_INDEX(lz_buff, lz_buff->offset - distance + i) ] !=
                lz_buff->buffer[ WRAP_BUFFER_INDEX(lz_buff, lz_buff->offset + i) ] ) {
            errcnt++;
            return 0;
        }
    return 1;
}


static size_t lz_encode_buffer(struct lz_buffer* lz_buff,struct lz_extra_data* extra_datas,uint16_t*
 lz_out_buffer, size_t* len)
{
    ssize_t distance;
    ssize_t length;
    size_t i;
    uint8_t last=0;
    size_t extra_datas_cnt = 0;


/* for(i=0;i<*len;i++)
        lz_out_buffer[i] = lz_buff->buffer[WRAP_BUFFER_INDEX(lz_buff, lz_buff->offset+i)];
    return 0;*/
    for(i=0;i < *len; i++) {
        const char c = lz_buff->buffer[WRAP_BUFFER_INDEX(lz_buff, lz_buff->offset)];
        lzbuff_search_longest_match(lz_buff, lz_buff->offset, lz_buff->buffer_len/2, &distance,
         &length);

        /*show_match(lz_buff,distance,length);*/


        /*show_lz_buff(lz_buff);*/


        if(length >= code_to_length[0].start && distance >0 && distance < lz_buff->buffer_len/2
         && check_lz(lz_buff,length,distance) ) {
            struct lz_extra_data* extra_data = &extra_datas[extra_datas_cnt++];


            extra_data->distance = distance;
```

```
            lz_out_buffer[i] = 0x100 + length_to_code(length, &extra_data->extra_bits, &extra_data
              ->extra_data);
            lz_buff->offset = WRAP_BUFFER_INDEX(lz_buff, lz_buff->offset + length);
            *len -= length-1;
        }
        else {
            lzbuff_insert(lz_buff, c);
            lz_out_buffer[i] = c;
        }
    }

    for(i=0;i<lz_buff->buffer_len/2;i++)
        lz_remove(lz_buff, lz_buff->offset+lz_buff->buffer_len/2+i);
    return extra_datas_cnt;
}

/* count number of occurances of each byte */
static void countblock(uint16_t *buffer, freq length, freq *counters)
{
     size_t i;
     /* zero counters */
     memset(counters, 0 ,sizeof(counters[0])*(SYMBOLS+1));
     /* then count the number of occurances of each byte */
     for (i=0; i<length; i++)
         counters[buffer[i]]++;
}
#define MIN(a,b) ((a)<(b)?(a):(b))
int main( int argc, char *argv[] )
{
    size_t blocksize;
    rangecoder rc;
    unsigned char buffer[BLOCKSIZE];
    uint16_t lz_out_buffer[BLOCKSIZE];
    struct lz_extra_data extra_datas[BLOCKSIZE];
    size_t extra_data_cnt = 0;
    size_t j;
    struct lz_buffer lz_buff;
    struct ari_model model;
    size_t processed = 0;

    if ((argc > 3) || ((argc>1) && (argv[1][0]=='-')))
        usage();

    if ( argc <= 1 )
        fprintf( stderr, "stdin" );
    else {
        freopen( argv[1], "rb", stdin );
        fprintf( stderr, "%s", argv[1] );
    }
    if ( argc <= 2 )
        fprintf( stderr, " to stdout\n" );
    else {
        freopen( argv[2], "wb", stdout );
        fprintf( stderr, " to %s\n", argv[2] );
```

```
    }

#ifndef unix
    setmode( fileno( stdin ), O_BINARY );
    setmode( fileno( stdout ), O_BINARY );
#endif

    /* initialize the range coder, first byte 0, no header */
    start_encoding(&rc,0,0);
    setup_lz_buffer(&lz_buff, 1+BLOCKSIZE_POWER);
    model_setup(&model);

    while (1)
    {
        freq i;
        size_t extra_datas_cnt;

        blocksize = MIN( BLOCKSIZE, lz_buff.buffer_len - lz_buff.offset);
        /* get the statistics */
        blocksize = fread(lz_buff.buffer+lz_buff.offset,1,(size_t)blocksize,stdin);

        /* terminate if no more data */
        if (blocksize==0) break;

        encode_freq(&rc,1,1,2); /* a stupid way to code a bit */
        /* blocksize, can be max 2^22, since we would need to restart coder anyway on <2^23*/
        extra_datas_cnt = lz_encode_buffer(&lz_buff,extra_datas,lz_out_buffer,&blocksize);

        encode_short(&rc, blocksize&0xffff);
        encode_short(&rc, blocksize>>16);
        /*countblock(lz_out_buffer,blocksize,counts);*/

        /* write the statistics. */
        /* Cant use putchar or other since we are after start of the rangecoder */
        /* as you can see the rangecoder doesn't care where probabilities come */
        /* from, it uses a flat distribution of 0..0xffff in encode_short. */

        /*fprintf(stderr,"Counters:\n");
          for(i=0;i<SYMBOLS;i++) {
          fprintf(stderr,"%d:%ld\n",i,counts[i]);
          }
          fprintf(stderr,"\n");*/

        /*for(i=0; i < SYMBOLS; i++)
            encode_short(&rc,counts[i]);*/

        /* store in counters[i] the number of all bytes < i, so sum up */
        /*counts[SYMBOLS] = blocksize;
        for (i=SYMBOLS; i; i--)
            counts[i-1] = counts[i]-counts[i-1];
         */
```

```
        /*
            fprintf(stderr,"Counters:\n");
            for(i=0;i<SYMBOLS;i++) {
            fprintf(stderr,"C%d:%ld\n",i,counts[i]);
            }
            fprintf(stderr,"\n");
         */

        /* output the encoded symbols */
        for(i=0,j=0; i<blocksize; i++) {
            freq cur_freq, cum_freq, total_freq;
            const uint16_t ch = lz_out_buffer[i];
            /*fprintf(stderr,"Encoding:%d,(%c),%ld,%ld\n",ch,ch,counts[ch+1]-counts[ch],counts[ch]);*/
            model_get_freq(&model,ch,&cur_freq,&cum_freq,&total_freq);
            encode_freq(&rc,cur_freq,cum_freq,total_freq);
            model_update_freq(&model,ch);
            if(ch > 0xff) {
                /* output length,distance */
                const struct lz_extra_data* extra_data = &extra_datas[j++];
                if(extra_data->extra_bits) {
                    encode_shift(&rc, (freq)1, (freq)extra_data->extra_data, extra_data->extra_bits);
                }
                    encode_shift(&rc,(freq)1,(freq)(extra_data->distance&0xff), 8);
                encode_shift(&rc,(freq)1,(freq)(extra_data->distance>>8),8);
            }
        }
        processed += blocksize;
        if(processed > 1<<19) {
            /* restart encoder */
            done_encoding(&rc);
            start_encoding(&rc,0,0);
            processed=0;
        }
        fflush(stdout);
        /*fprintf(stderr,"%d\n",model.counts[SYMBOLS]);*/
    }

    /* flag absence of next block by a bit */
    encode_freq(&rc,1,0,2);

    /* close the encoder */
    done_encoding(&rc);
    model_done(&model);
    cleanup_lz_buffer(&lz_buff);

    fprintf(stderr,"Missed: %ld LZ77 encoding opportunities\n",errcnt);
    fprintf(stderr,"Processed:%ld bytes\n",processed);
    return 0;
}
```

A.11 range_decod.h

```
/*
 * Singularity-compress: rangecoder decoder
```

```
* Copyright (C) 2006–2007    Torok Edwin (edwintorok@gmail.com)
*
* Based on Michael Schindler's rangecoder, which is:
* (c) Michael Schindler 1997, 1998, 1999, 2000
* http://www.compressconsult.com/
*
*
* This program is free software; you can redistribute it and/or
* modify it under the terms of the GNU General Public License
* as published by the Free Software Foundation; version 2
* of the License.

* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.

* You should have received a copy of the GNU General Public License
* along with this program; if not, write to the Free Software
* Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA          02110–1301, USA.
*/

#ifndef _RANGE_DECOD_H
#define _RANGE_DECOD_H

#ifndef NDEBUG
#define DO_CHEKCS
#endif

#include "../common/rangecod.h"

/* Start the decoder                                                    */
/* rc is the range coder to be used                                     */
/* returns the char from start_encoding or EOF                          */
static inline int start_decoding( rangecoder *rc )
{
    int c = inbyte(rc);
    if (c==EOF)
        return EOF;
    rc->buffer = inbyte(rc);
    rc->low = rc->buffer >> (8–EXTRA_BITS);
    rc->range = (code_value)1 << EXTRA_BITS;
    return c;
}

static inline void dec_normalize( rangecoder *rc )
{
    while (rc->range <= Bottom_value)     {
        rc->low = (rc->low<<8) | ((rc->buffer<<EXTRA_BITS)&0xff);
        rc->buffer = inbyte(rc);
        rc->low |= rc->buffer >> (8–EXTRA_BITS);
        rc->range <<= 8;
    }
}
```

```
/* Calculate culmulative frequency for next symbol. Does NO update!*/
/* rc is the range coder to be used                                 */
/* tot_f is the total frequency                                     */
/* or: totf is (code_value)1<<shift                                 */
/* returns the culmulative frequency                                */
static inline freq decode_culfreq( rangecoder *rc, freq tot_f )
{
    freq tmp;
    dec_normalize(rc);
    rc->help = rc->range/tot_f;
    tmp = rc->low/rc->help;
    return (tmp>=tot_f ? tot_f-1 : tmp);
}



static inline freq decode_culshift( rangecoder *rc, freq shift )
{
    freq tmp;
    dec_normalize(rc);
    rc->help = rc->range>>shift;
        tmp = rc->low/rc->help;
    return (tmp>>shift ? ((code_value)1<<shift)-1 : tmp);
}


/* Update decoding state                                            */
/* rc is the range coder to be used                                 */
/* sy_f is the interval length (frequency of the symbol)            */
/* lt_f is the lower end (frequency sum of < symbols)               */
/* tot_f is the total interval length (total frequency sum)         */
static inline void decode_update( rangecoder *rc, freq sy_f, freq lt_f, freq tot_f)
{
    code_value tmp;
    tmp = rc->help * lt_f;
    rc->low -= tmp;
    if (lt_f + sy_f < tot_f)
        rc->range = rc->help * sy_f;
    else
        rc->range -= tmp;
}

#define decode_update_shift(rc,f1,f2,f3) decode_update((rc),(f1),(f2),(freq)1<<(f3));

/* Decode a byte/short without modelling                            */
/* rc is the range coder to be used                                 */
static inline unsigned char decode_byte(rangecoder *rc)
{
    unsigned char tmp = decode_culshift(rc,8);
    decode_update( rc,1,tmp,(freq)1<<8);
    return tmp;
}
```

```
static inline unsigned short decode_short(rangecoder *rc)
{
    unsigned short tmp = decode_culshift(rc,16);
    decode_update( rc,1,tmp,(freq)1<<16);
    return tmp;
}

/* Finish decoding                                              */
/* rc is the range coder to be used                             */
static inline void done_decoding( rangecoder *rc )
{
    dec_normalize(rc);          /* normalize to use up all bytes */
}

#endif
```

A.12 simple_d.c

```
/*
 * Singularity–compress: arithmetic decoder, and lz77 decoder
 *
 * Based on Michael Schindler's rangecoder, which is:
 * (c) Michael Schindler 1999
 * http://www.compressconsult.com/
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; version 2
 * of the License.

 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.

 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110–1301, USA.
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>

#include "../common/code_tables.h"

#include "range_decod.h"

#include "../common/model.h"
```

```
struct buffer

    unsigned char* data;
    size_t         len;
    ssize_t        len_mask;
    size_t         len_power;
    size_t         offset;
};

#define MIN(a,b) ((a)<(b) ? (a) : (b))

static ssize_t copy_back_bytes(struct buffer* buff,ssize_t dest,ssize_t backbytes,size_t backsize)
{
    size_t from = (dest - backbytes) & buff->len_mask;

    size_t dest_maxcopy = MIN(buff->len - dest, backsize);
    size_t src_maxcopy =  MIN(buff->len - from, backsize);

    if(src_maxcopy < dest_maxcopy) {
        memcpy(&buff->data[dest], &buff->data[from], src_maxcopy);

        dest += src_maxcopy;
        backsize -= src_maxcopy;
        from  = (from + src_maxcopy)&buff->len_mask;
        dest_maxcopy -= src_maxcopy;

        if(dest_maxcopy)

            memcpy(&buff->data[dest], &buff->data[from], dest_maxcopy);

            from += dest_maxcopy;
            backsize -= dest_maxcopy;
            dest  = (dest + dest_maxcopy)&buff->len_mask;
                if(!dest)
               fwrite(buff->data,1,buff->len,stdout);

            memcpy(&buff->data[dest], &buff->data[from], backsize);
            dest += backsize;
        }
    }
    else

        memcpy(&buff->data[dest], &buff->data[from], dest_maxcopy);

        from += dest_maxcopy;
        backsize -= dest_maxcopy;
        dest = (dest + dest_maxcopy)&buff->len_mask;
        src_maxcopy -= dest_maxcopy;

        if(!dest)
            fwrite(buff->data,1,buff->len,stdout);
        if(src_maxcopy)
```

```
            memcpy(&buff->data[dest], &buff->data[from], src_maxcopy);

            dest += src_maxcopy;
            backsize -= src_maxcopy;
            from = (from + src_maxcopy) &buff->len_mask;

            memcpy(&buff->data[dest],&buff->data[from], backsize);
            dest += backsize;
        }
    }
    return dest;
}



int unpack(FILE* out,FILE* in)
{
    freq cf;
    rangecoder rc;
    struct buffer buffer;
    struct ari_model model;
    size_t processed=0;

    buffer.offset = 0;
    buffer.len_power = 1+BLOCKSIZE_POWER;
    buffer.len = 1<<buffer.len_power;
    buffer.len_mask = buffer.len - 1;
    buffer.data = malloc(buffer.len);
    rc.in = in;
    model_setup(&model);

    if(!buffer.data)
        return -2;

    if (start_decoding(&rc) != 0) {
        return -1;
    }

    while ( (cf = decode_culfreq(&rc,2)) ) {
        freq i, blocksize;

        decode_update(&rc,1,1,2);

        blocksize = decode_short(&rc) | ((size_t)decode_short(&rc)) <<16;

        for (i=0; i<blocksize; i++) {
            freq symbol;

            cf = decode_culfreq(&rc,model.counts[SYMBOLS]);

            symbol = model_get_symbol(&model, cf);
            decode_update(&rc, model.counts[symbol+1]-model.counts[symbol],model.counts[symbol],
```