Parallel Text Compression

Source

Honours Project 1989

Craig Nevill
Supervisor Tim Bell

Department of Computer Science
University of Canterbury

main.c for parallel compressor

```
#include <stdio.h>
#include <math.h>
#define mainprog true
#include "ppm.h"
#include "pc.h"
#define buffsize    2048
#define blocksize   (buffsize*10)
#define shift_st    for (i=1;i<=maxorder;i++) s[i] = t[i-1]
#define max(a, b) ((a) > (b) ? (a) : (b))

eventnode *newnode() ;
void rebuild_tree();
void arithmetic_encode();
void doneencoding();
void startoutputingbits();
void startencoding();
void write_method();
 void doneoutputingbits();
void update_ppm_model();
void build_ppm_distribution();
void prime();
extern unsigned char charbuff[buffsize] ;
extern int chp;
double logs[16384];
float parallel_likely[nchars], ppm_likely[nchars], weight[11];
float ppm_confidence, parallel_confidence, real_combined[nchars +
1];
int eoverse = 0, eof = 0, combined[nchars + 1];
int nothing_weight, syn_weight, word_weight, eol_weight,
syn_threshold;
FILE *source;

main(argc, argv)
int argc;
char *argv[];
{
  event e ;
  int very_first_time = 1, encoding = 1;
  char ch;

  if (argc > 10)
  {
    int i;
    for (i = 2; i < 13; i ++)
       weight[i - 2] = atoi(argv[i]) / (float) 100;
     nothing_weight = atoi(argv[13]);
     word_weight = atoi(argv[14]);
     syn_weight = 1000 - word_weight;
     eol_weight = atoi(argv[15]);
     syn_threshold = atoi(argv[16]);
  }
  else
  {
    fprintf(stderr, "-e/d,   10 * paralllel weights, nothing weight,
\
 word_weight(%), eol weight, syn threshold\n");
    exit(1);
  }
  if ((source = fopen("source", "r")) == NULL)
  {
     fprintf(stderr, "Couldn't open source version\n");
     exit(1);
  }

  if (argc > 1 && argv[1][0] == '-' && argv[1][1] == 'd')
    encoding = 0;

  E = (eventptr) calloc(kbytes, 1024) ;

  if (encoding)
  {
     startoutputingbits() ;
     startencoding() ;
  }
  else
  {
     startinputingbits() ;
     startdecoding() ;
  }

/*   calculate_logs();*/

  while (!eof)
  {
     read_source();
     reset_parallel();
     if (very_first_time)
     {
        prime(true, encoding) ;
        very_first_time = 0;
     }
     eoverse = 0;
     while (!eoverse)
     {
        build_ppm_distribution(3);
        build_parallel_distribution();
        combine_distributions();
        eof = endecode(&e, encoding);
        charbuff[chp = (chp+1) % buffsize] = e ;
        ch = event_to_char(e);
```

```
      eoverse = e == end_of_verse;
      update_ppm_model(3, e);
      update_parallel_model(ch);
      if (nfreenodes<=order)
      {
        rebuild_tree() ;
      }
    }
  }
  if (encoding)
  {
    arithmetic_encode(16381, 16383, 16383);
    doneencoding() ;
    doneoutputingbits() ;
  }
}

combine_distributions()
{
  int i, rescale;
  float parallel_ent = 0.0, ppm_ent = 0.0;
  float lbnd = 0.0, hbnd, total, code_space;

  rescale = 0;
  combined[0] = 0;
  /*
  for (i = 0, lbnd = 0.0; i < nchars; i ++)
  {
    parallel_ent += parallel_likely[i] * logs[(int)
(parallel_likely[i] * 16381) / 8];
    ppm_ent += ppm_likely[i] * logs[(int) (ppm_likely[i] * 16381)
/ 8];
  }
  fprintf(stderr, "Entropy: ppm %f parallel %f\n", ppm_ent,
parallel_ent);
  */
  for (i = 0, lbnd = 0.0; i < nchars; i ++)
  {
    code_space = (ppm_likely[i] * (1 - parallel_confidence)
                + parallel_likely[i] * parallel_confidence);

    if (16381 * code_space < 1.0)
    {
      rescale = 1;
      lbnd += (float) 2 / 16381;
    }
    else
      lbnd += code_space;
    combined[i + 1] = lbnd * 16381;
    real_combined[i + 1] = lbnd; /* For rescaling */
  }


  combined[nchars] = 16381;
  if (rescale)
  {
    for (i = 0, total = 0.0; i < nchars; i ++)
      combined[i + 1] = real_combined[i + 1] / lbnd * 16381;
  }
/*  for (i = 0, totl = 0.0; i < nchars; i ++)
    totl += max(ppm_likely[i], parallel_likely[i] *
parallel_confidence);
  */
}

int endecode(e, encoding)
event *e;
int encoding;
{
  int target, i;
  float float_target;
  char ch;

  if (encoding)
  {
    ch = getchar();
    if (ch == EOF)
      return 1;
    *e = char_to_event(ch);
    charbuff[chp] = *e;
    rawbytes ++ ;
#ifdef DEBUG
  fprintf(stderr, "%c %d %f %f %f %f %d %d\n", event_to_char(*e),
*e,
  ppm_likely[*e], parallel_likely[*e],
  (ppm_likely[*e]+ parallel_likely[*e] * parallel_confidence)
/ (1 + parallel_confidence),
   parallel_confidence, combined[*e], combined[*e + 1]);
   fprintf(stderr, "Bits = %g\n", logs[(combined[*e + 1] -
combined[*e]]) / 8);

#endif
    arithmetic_encode(combined[*e], combined[*e + 1], 16383);
  }
  else
  {
    target = arithmetic_decode_target(16383);
    if (target > 16381)
    {
      eoverse = 1;
```

```
      return true;
    }
    else
    {
      for (i = 0; target >= combined[i] && i < nchars; i ++)
        ;
      *e = i - 1;
      putchar(event_to_char(*e));
      fflush(stdout);
      arithmetic_decode(combined[*e], combined[*e + 1], 16383);
#ifdef DEBUG
  fprintf(stderr, "%c %d %f %f %f %f %d %d\n", event_to_char(*e),
*e,
  ppm_likely[*e], parallel_likely[*e],
  (ppm_likely[*e]+ parallel_likely[*e] * parallel_confidence)
/ (1 + parallel_confidence),
   parallel_confidence, combined[*e], combined[*e + 1]);
#endif
    }
  }
  return 0;
}


void prime(coding, encoding)
boolean coding;
int encoding ;
{ register int j ;
  event e ;

  numnodes = 0 ;
  nfreenodes = maxnodes ;
  s[0] = newnode(NULL) ;

  /* prime the pump as many times as necessary */
  for (j=0; j<order; j++)
  {
    chp = (chp+1) % buffsize ;
    if (coding)
    {
      build_ppm_distribution();
      build_parallel_distribution();
      combine_distributions();
      endecode(&e, encoding);
    }
    else
      e = charbuff[chp] ;
    update_ppm_model(j, e);
    update_parallel_model(event_to_char(e));
  }
}

calculate_logs()
{
  int i;

  for (i = 8; i < 16384; i += 8)
  {
    logs[i / 8] = -(double) log2((double) i / 16382);
  }
  logs[0] = 0;
}
```

pc.h for parallel compressor


```
#define abs(a) ((a) > 0 ? (a) : -(a))
#define BUCKETS 29999

char self[BUCKETS];

struct syn_used
{
   char pos_no1, pos_no2, syn_no1, syn_no2, count1, count2;
} synonyms[BUCKETS];

struct pos
{
   char pos[5];
   char syn[200][16];
   int  no_syn;
};

struct word
{
   char word[17];
   struct pos posse[6];
   int no_pos;
   char *syn1, *syn2;
   char count1, count2;
   char self;
} verse[200];

struct syn_used *synonyms_used;
```

parallel.c for P.C.

```
#include <stdio.h>
#include "ppm.h"
#include "pc.h"

extern float parallel_confidence, parallel_likely[nchars],
weight[11];
extern FILE *source;

extern int nothing_weight, syn_weight, word_weight, eol_weight,
syn_threshold;
int i, j, eoverse, eolist, eopos, no_words, pos_no, likelyhood,
likely[nchars],
     syn_no, w, p, syn_no, position, sure_of_pos, words_matched,
syn_matched, total;
char words[20][100], so_far[20];

int total_target_words = 0, total_source_words = 0,
     words_expected = 0, target_words = 0;

struct syn_used *find_syn();
char find_self();

read_source()
{

  char ch;
  eoverse = 0;
  i = 0;
  no_words = 0;
  while (!eoverse)
  {
    eolist = 0;
     fscanf(source, "%[a-z]", verse[no_words].word);
     if (strcmp(verse[no_words].word, "x") == 0)
    {
       eoverse = 1;
       getc(source); /* read \n*/
#ifdef PARSE_DEBUG
       printf("\n********************** End of verse
****************\n");
#endif
    }
    else
    {
#ifdef PARSE_DEBUG
         printf("\nword: %s", verse[no_words].word);
#endif
       synonyms_used = find_syn (verse[no_words].word);
       if (synonyms_used->count1 == 0)
      {
         verse[no_words].syn1 = NULL;
         verse[no_words].syn2 = NULL;
       }
       else
       {
         verse[no_words].syn1 =
          verse[no_words].posse[synonyms_used-
>pos_no1].syn[synonyms_used->syn_no1];
         verse[no_words].syn2 =
          verse[no_words].posse[synonyms_used-
>pos_no2].syn[synonyms_used->syn_no2];
          verse[no_words].count1 = synonyms_used->count1;
          verse[no_words].count2 = synonyms_used->count2;
          verse[no_words].self = find_self(verse[no_words].word);
       }
       pos_no = 0;
       if (getc(source) != '\n')
         while (!eolist)
         {
            ch = getc(source);
            if (ch == '~')
            {
              eolist = 1;
               getc(source); /* read the \n */
             }
            else
            {
              if (ch != '|')
              {
                return;
    /*            fprintf(stderr, "Parse error: missing bar\n");
*/
              }
               fscanf(source, "%[a-z]",
verse[no_words].posse[pos_no].pos);
/*               printf("\npos: %s",
verse[no_words].posse[pos_no].pos);*/

             if (getc(source) != ':')
                fprintf(stderr, "Parse error: missing colon\n");
             eopos = 0;
             syn_no = 0;
             while (!eopos)
             {
                if (fscanf(source, "%[a-z' -]",
                           verse[no_words].posse[pos_no].syn[syn_no])
== 0)
                  eopos = 1;
                else
                {
```

```
/*                  printf(" %s",
verse[no_words].posse[pos_no].syn[syn_no]);*/
                ch = getc(source);
                if (ch != ',')
                  fprintf(stderr, "Parse error: missing comma
\n");
              }
              syn_no ++;
            } /* while !eopos */
            verse[no_words].posse[pos_no].no_syn = syn_no;
            pos_no ++;
          }
        } /* while !eolist */
        verse[no_words].no_pos = pos_no;
        no_words ++;
      }
    } /* while !eopos */
    if (words_expected == 0)
      words_expected = no_words;
    else
      words_expected = no_words * total_target_words /
total_source_words;

}

reset_parallel()
{
  words_matched = i = sure_of_pos = 0;
  i = 0;
  eoverse = 0;
  position = 0;
}

char last_word[30];
update_parallel_model(ch)
char ch;
{
  if (ch == ' ' || ch == '\n')
  {
    total_target_words ++;
    target_words ++;
    if (ch == '\n')
    {
      total_source_words += no_words;
#ifdef DEBUG
      fprintf(stderr, "Length difference = %d\n", target_words -
words_expected);
#endif
      target_words = 0;
    }
    parallel_confidence = 0.0;
    strncpy(last_word, so_far, i);
    last_word[i] = '\0';
#ifdef DEBUG
    fprintf(stderr, "Word was %s\n", last_word);
#endif
    i = 0;
    return;
  }
  so_far[i ++] = ch;
}

build_parallel_distribution()
{

  for (j = 0; j < nchars; j ++)
    likely[j] = nothing_weight;

  if (i == 0)
  {
    find_position();
    for (w = 0; w < no_words; w ++)
    {
      likelyhood = (verse[w].self == 0 ? 1 : verse[w].self)
    * (word_weight - word_weight * abs(w - position) * sure_of_pos
                        / (no_words * 100));
      likely[verse[w].word[i] - 'a'] += likelyhood;
    }
    words_matched = 1;
  }
  else
  {
    words_matched = syn_matched = 0;

  /*          printf("%.*s ", i, so_far);*/
    for (w = 0; w < no_words; w ++)
    {
      if (strncmp(so_far, verse[w].word, i) == 0)
      {
  #ifdef DEBUG
        fprintf(stderr, "Word: %s\n", verse[w].word);
  #endif
        likelyhood = (verse[w].self == 0 ? 1 : verse[w].self)
      * (word_weight - word_weight * abs(w - position) *
sure_of_pos
                          / (no_words * 100));
        if (i == strlen(verse[w].word))
          likely[space] += likelyhood;
        else
          likely[verse[w].word[i] - 'a'] += likelyhood;
```

```
        words_matched ++;
      }

      if (verse[w].syn1 != NULL && strncmp(so_far, verse[w].syn1,
i) == 0)
      {
  #ifdef DEBUG
        fprintf(stderr, "Synonym 1: %s\n", verse[w].syn1);
  #endif
        likelyhood = verse[w].count1 * (word_weight - word_weight
* abs(w - position) * sure_of_pos
                        / (no_words * 100));
        if (i == strlen(verse[w].syn1))
          likely[space] += likelyhood;
        else
          likely[verse[w].word[i] - 'a'] += likelyhood;
        words_matched ++;
      }

      if (verse[w].syn2 != NULL && strncmp(so_far, verse[w].syn2,
i) == 0)
      {
  #ifdef DEBUG
        fprintf(stderr, "Synonym 2: %s\n", verse[w].syn2);
  #endif
        likelyhood = verse[w].count2 * (word_weight - word_weight
* abs(w - position) * sure_of_pos
                        / (no_words * 100));
        if (i == strlen(verse[w].syn2))
          likely[space] += likelyhood;
        else
          likely[verse[w].word[i] - 'a'] += likelyhood;
        words_matched ++;
      }


    }
    if (words_matched == 0 && i > syn_threshold)
    for (w = 0; w < no_words; w ++)
      for (p = 0; p < verse[w].no_pos; p ++)
      {
        for (syn_no= 0; syn_no< verse[w].posse[p].no_syn;
syn_no++)
        {
          if (strncmp(so_far, verse[w].posse[p].syn[syn_no], i) ==
0)
          {
  #ifdef DEBUG
            fprintf(stderr, "Synonym: %s %d %.*s\n",
verse[w].posse[p].syn[syn_no], i, i, so_far);
  #endif

            likelyhood = i * (syn_weight - syn_weight * abs(w -
position) * sure_of_pos
                    / (100 * no_words));
            if (i == strlen(verse[w].posse[p].syn[syn_no])
             || verse[w].posse[p].syn[syn_no][i] < 'a'
             || verse[w].posse[p].syn[syn_no][i] > 'z')
              likely[space] += likelyhood;
            else
              likely[verse[w].posse[p].syn[syn_no][i] - 'a'] +=
likelyhood;
            syn_matched ++;
          }
        }
      }
    } /* if i != 0 */
  /*      putchar('\n');*/
    if (syn_matched == 0 && words_matched == 0)
      parallel_confidence = 0.0;
    else
    {
      if (i < 11)
        parallel_confidence = weight[i];
      else
        parallel_confidence = weight[10];
        /*= 4 * (1 - 0.5 / i);*/
  /*          printf("I am %d %% confident\n", confidence);*/

      total = 0;

      if (words_expected - target_words < 8)
      {
        int distance;
        distance = words_expected - target_words + 1;
        distance = distance < 1 ? 1 : distance;

        likely[end_of_verse] = likely[space] * eol_weight /
(distance * 100);
        likely[space] -= likely[end_of_verse];
      }

      for (j = 0; j < nchars; j ++)
        total += likely[j];
      if (total == 0)
        total = 1;

  #ifdef DEBUG
      fprintf(stderr, "\n%.*s ", i, so_far);
  #endif
```

```
      for (j = 0; j < nchars; j ++)
      {
  #ifdef DEBUG
        if (likely[j] > 1)
          fprintf(stderr, "%c %d  ", j + 'a', likely[j]);
  #endif
      parallel_likely[j] = (float) likely[j] / total;

      }
/*    putchar('\n');*/
    }
}

find_position()
{
  int length, words_matched = 0, syn_matched = 0, synonym, part;

  length = strlen(last_word); /* allows for suffixes */

  for (w = 0; w < no_words; w ++)
    if (strncmp(last_word, verse[w].word, length) == 0
    && (length > 2 || strlen(verse[w].word) < 5))
    {
      words_matched ++;
      position = w;
    }

  if (!words_matched && length > 2)
    for (w = 0; w < no_words; w ++)
      for (p = 0; p < verse[w].no_pos; p ++)
        for (syn_no= 0; syn_no< verse[w].posse[p].no_syn;
syn_no++)
          if (strncmp(last_word, verse[w].posse[p].syn[syn_no],
length) == 0
              && length == strlen(verse[w].posse[p].syn[syn_no]))
          {
            syn_matched ++;
            position = w;
            part = p;
           synonym = syn_no;
          }

  if (words_matched + syn_matched == 1)
  {
    sure_of_pos = 100;
    if (words_matched == 1)
      insert_self(last_word);
    else
      insert_syn(last_word, part, synonym);
  }
  else
  {
    sure_of_pos -= 20;
    sure_of_pos = sure_of_pos < 0 ? 0 : sure_of_pos;
  }
  position ++;

}
```

modified PPM code


```
/* PPM.C

    Multi order character PPM text compression, with exclusions.
    Alistair Moffat, July 1987, December 1987.
*/

#include <stdio.h>
#include <ctype.h>
#include "ppm.h"

extern float like[nchars];

eventnode *encode_event_noex() ;
eventnode *decode_event_noex() ;
eventnode *encode_event() ;
eventnode *decode_event() ;
eventnode *addevent() ;
eventnode *newnode() ;

void arithmetic_encode() ;
void arithmetic_decode() ;
void startoutputingbits() ;
void startencoding() ;
void doneencoding() ;
void doneoutputingbits() ;
void startinputingbits() ;
void startdecoding() ;
void update_ppm_model() ;

#define buffsize   2048
#define blocksize  (buffsize*10)
#define shift_st   for (i=1;i<=maxorder;i++) s[i] = t[i-1]

#define set_ex(p,e)                                  \
  do                                                 \
    { register eventptr q ;                          \
      for (q=p32(p); q!=ENULL; q=p32(q->rght))       \
      excluded[q->eventnum] = true ;                 \
      excluded[e] = false ;                          \
    }                                                \
  while (false)

#define clear_ex(p)                                  \
  do                                                 \
    { register eventptr q ;                          \
      for (q=p32(p); q!=ENULL; q=p32(q->rght))       \
      excluded[q->eventnum] = false ;                \
    }                                                \
  while (false)

unsigned char      charbuff[buffsize] ;
int          chp=buffsize-1 ;
long         lastbuiltat=0 ;

/*==================================*/

/*==================================*/


void update_ppm_model(n, e)
  int n ;
  event e ;
  { boolean encoded, coding = 0;
    int i, c ;
    float lbnd, hbnd;

    t[n] = encode_event_noex(&s[n]->foll, e, &encoded, coding) ;
    if (encoded)
      c = n ;
    else
      for (c=n-1; c>=0; c--)
      {
#ifdef noexclusions
        t[c] = encode_event_noex(&s[c]->foll, e, &encoded,
coding) ;
#else
        set_ex(s[c+1]->foll.tree, e) ;
          t[c] = encode_event(&s[c]->foll, e, &encoded, coding) ;
#endif
          if (encoded) break ;
      }
    if (!encoded && coding)
      { dummy_encode(e, e+1, nchars) ;
      }
    for (i=c+1; i<=n; i++)
      { if (i>0) t[i]->prev = p16(t[i-1]) ;
      else   t[0]->prev = p16(s[0]) ;
      }
    for (i=c-1; i>=0; i--)
      t[i] = p32(t[i+1]->prev) ;
#ifndef noexclusions
    if (c!=n)
      clear_ex(s[c+1]->foll.tree) ;
#endif
    shift_st ;
  }

/*==================================*/

/*void decode_inorder(n, e)
```

```
  int n ;
  event *e ;
  { boolean decoded ;
    int i, c ;
     t[n] = decode_event_noex(&s[n]->foll, &decoded, e) ;
    if (decoded)
      c = n ;
    else
       for (c=n-1; c>=0; c--)
       {
#ifdef  noexclusions
            t[c] = decode_event_noex(&s[c]->foll, &decoded, e) ;
#else
         set_ex(s[c+1]->foll.tree, EOFchar) ;
            t[c] = decode_event(&s[c]->foll, &decoded, e) ;
#endif
           if (decoded) break ;
       }
    if (!decoded)
       { *e = arithmetic_decode_target(nchars) ;
         arithmetic_decode(*e, *e+1, nchars) ;
      }
    for (i=c+1; i<=n; i++)
      { t[i]->eventnum = *e ;
        if (i>0) t[i]->prev = p16(t[i-1]) ;
      else    t[0]->prev = p16(s[0]) ;
      }
    { register int i ;
      for (i=c-1; i>=0; i--)
      t[i] = p32(t[i+1]->prev) ;
    }
#ifndef  noexclusions
    if (c!=n)
       clear_ex(s[c+1]->foll.tree) ;
#endif
    shift_st ;
  }
*/
/*==================================*/


void rebuild_tree()
  { int chrs ;
    fprintf(stderr, "Rebuilding Tree\n");
#ifdef  decreasingorder
     if (rawbytes-lastbuiltat<blocksize)
      order -= 1 ;
#endif
     lastbuiltat = rawbytes ;
     if (rawbytes<buffsize)
       { /* buffer is not even full yet */
       chp = buffsize-1 ;
       chrs = rawbytes ;
       }
     else
       chrs = buffsize ;
     prime(false) ;
     for (chrs = chrs-order; chrs>0; chrs--)
       { chp = (chp+1) % buffsize ;
       update_ppm_model(order, (event)charbuff[chp]) ;
       }
     if (nfreenodes<maxnodes/5)
       { fprintf(stderr, "Insufficient space to rebuild trie\n") ;
       exit(-1) ;
       }
  }

/*==================================*/
```

wf.c - word finder

```
#include <stdio.h>

#define BUCKETS      29999
#define NOUN         0x0
#define VERB         0x1
#define ADJECTIVE    0x2
#define ADVERB       0x3
#define PRONOUN      0x4
#define CONJUNCTION  0x5
#define ARTICLE      0x6
#define PREPOSITION  0x7

struct word *find_or_insert();
struct word *find();


struct syn_list
{
   struct word **list;
   struct syn_list *next;
} list_supply[32000];

struct word
{
   char *word;
   struct syn_list *synonyms;
   struct word *next;
} word_supply[15000];

char string_supply[140000];
int chars = 0;

struct word *thesaurus[BUCKETS], *words[100];
int lines = 0;

main()
{
   char word[100], pos[10], pos_number, *part[10], ch,
string[1000];
   int done, wc, i, chars, j, first_time, wds = 0;
   struct word **list_head, **syn_ptr, *this_syn;
   struct syn_list *temp_list;
   struct word *word_ptr;
   struct syn_list *list_ptr;
   FILE *Large_Thesaurus;


   Large_Thesaurus = fopen("/usr/local/lib/thesaurus", "r");
   part[0] = "noun";
   part[1] = "verb";
   part[2] = "adj";
   part[3] = "adv";
   part[4] = "pron";
   part[5] = "conj";
   part[7] = "prep";

   for (i = 0; i < BUCKETS; i ++)
     thesaurus[i] = NULL;

   while (!feof(Large_Thesaurus))
   {
      fscanf(Large_Thesaurus, "%s:\t", pos);

     /* find part of speech */
     if (pos[0] == 'n')
       pos_number = NOUN;
     else if (pos[0] == 'v')
       pos_number = VERB;
     else if (pos[0] == 'p')
       pos_number = PREPOSITION;
     else if (pos[0] == 'c')
       pos_number = CONJUNCTION;
     else if (pos[0] == 'a')
       if (strncmp(pos, "adv", 3) == 0)
         pos_number = ADVERB;
       else
         pos_number = ADJECTIVE;

     done = wc = 0;

     while (!done)
     {
        ch = fgetc(Large_Thesaurus);
       i = 0;
       /* get a word */
        while ((word[i] = fgetc(Large_Thesaurus)) != ','
             && word[i] != '\n'
              && !feof(Large_Thesaurus))
         i ++;

        done = (word[i] == '\n') || feof(Large_Thesaurus);
       word[i] = '\0';
       /* find or insert the word and put the pointer in words */
        words[wc] = find_or_insert(word);
       wc ++;
     }

      list_head = (struct word **)
                   malloc((unsigned) ((wc + 1) * sizeof(struct word
*)));
```

```
    for (i = 0; i < wc; i ++)
      list_head[i] = words[i];

    /* store part of speech in top 8 bits of the first word
pointer
       in the list */
    list_head[0] = (struct word *) ((pos_number << 24) | (int)
list_head[0]);

    list_head[i] = NULL;

    /* point pointers from each word in the line to the array of
pointers */
    for (i = 0; i < wc; i ++)
    {
      temp_list = words[i]->synonyms;
      words[i]->synonyms = &list_supply[wds ++];
      words[i]->synonyms->next = temp_list;
      words[i]->synonyms->list = list_head;
    }
  }

  fprintf(stderr, "Finished reading thesaurus\n");

  while (!feof(stdin))
  {
    if (scanf("%s", word) != EOF)
    {
      word_ptr = find(word);
      if (word_ptr == NULL)
        printf("\n%s", word);
      else
      {
        printf("\n%s ", word);
        for (j = 0; j <= PREPOSITION; j ++)
        {
          first_time = 1;
          for (list_ptr = word_ptr->synonyms;
               list_ptr != NULL;
               list_ptr = list_ptr->next)
            if (((int) list_ptr->list[0] >> 24) == j)
            {
              if (first_time)
              {
                printf("|%s:", part[(int) list_ptr->list[0] >>
24]);
                first_time = 0;
              }
              chars = 8;
              for (syn_ptr = list_ptr->list, i = 0; syn_ptr[i] !=
NULL; i ++)
              {
                if (i == 0)
                  this_syn = (struct word *) ((int)syn_ptr[i] &
0xffffff);
                else
                  this_syn = syn_ptr[i];
                if (this_syn != word_ptr)
                {
                  printf("%s", this_syn->word);
                  printf(",");
                }
              }
            }
          }
          putchar('~');
        }
        if (getchar() == '\n')
          printf("\nx");
      }
    }
    putchar('\n');
}


/* hash the word into an array of pointers; handle collisions by
linked lists
from array element, create a new node if not already there */

struct word *find_or_insert(this_word)
char *this_word;
{
  int len;
  unsigned long hash_value = 0;
  char *word_temp;
  struct word *word_ptr;

  word_temp = this_word;
  for (len = strlen(this_word); len != 0; len --)
  {
    hash_value *= 4;
    hash_value += *(word_temp ++);
   }
  hash_value %= BUCKETS;

  if (thesaurus[hash_value] != NULL)
  {
   for (word_ptr = thesaurus[hash_value];
```

```
      strcmp(word_ptr->word, this_word) != 0 && word_ptr->next !=
NULL;
      word_ptr = word_ptr->next)
    ;

    if (strcmp(word_ptr->word, this_word) == 0)
      return word_ptr;

    word_ptr->next = &word_supply[lines ++];
    word_ptr = word_ptr->next;
  }
  else
  {
    thesaurus[hash_value] = &word_supply[lines ++];
    word_ptr = thesaurus[hash_value];
  }

  word_ptr->word = &string_supply[chars];
  chars += strlen(this_word) + 1;

  strcpy(word_ptr->word, this_word);
  word_ptr->synonyms = NULL;
  word_ptr->next = NULL;

  return(word_ptr);
}

struct word *find(this_word)
char *this_word;
{
  int len;
  unsigned long hash_value = 0;
  char *word_temp;
  struct word *word_ptr;

  word_temp = this_word;
  for (len = strlen(this_word); len != 0; len --)
  {
    hash_value *= 4;
    hash_value += *(word_temp ++);
   }
  hash_value %= BUCKETS;


  if (thesaurus[hash_value] != NULL)
  {
    for (word_ptr = thesaurus[hash_value];
         strcmp(word_ptr->word, this_word) != 0 && word_ptr->next
!= NULL;
         word_ptr = word_ptr->next)
      ;

    if (strcmp(word_ptr->word, this_word) == 0)
      return word_ptr;
    else
      return NULL;
  }
  else
    return NULL;
}
```

newprobc – modifies betting probabilities of subjects SE Appendix A

```
#include <stdio.h>
#include <ctype.h>

#define abs(a) ((a) > 0 ? (a) : -(a))
#define MAX_PERCENT 0.99

char results[5000][100];
struct char_info
{
   char character;
   float probability,
         bits;
   int time_taken, attempts;
} info[1000];

  float total = 1;
main()
{
   int length, wins;

   length = read_results();
   extract_info(length);
}

read_results()
{
   int line;
   char ch = '\0', character;

   for (line = 0; ch != EOF; line ++)
      for (character = 0; ((ch = getchar()) != '\n') && (ch != EOF)
; character ++)
         if (ch == '\n')
            results[line][character] = '\0';
         else
            results[line][character] = ch;
   return (line);
}

extract_info(length)
int length;
{
   int line, last_win = 0, characters_available, characters_chosen,
   hours, minutes, seconds, now, last_time, wins = 0, times;

   char characters[100], right_letter[2];
   float percentage, orig_percentage, percentage_so_far,
norm_percentage,
         original_psf;
   float certain, very_sure, quite_sure, probably, likely,
possibly,
         conceivably, no_idea;
   float certain_distance, very_sure_distance, quite_sure_distance,
    probably_distance, likely_distance, possibly_distance,
         conceivably_distance, no_idea_distance;
   float breakeven;

   printf("%s\n", results[0]);

   for (line = 1; line < length; )
   {
      for (; in("Skip", results[line]); line ++)
        printf("%s\n", results[line]);
      printf("%s\n", results[line]);

        sscanf(results[line], "At %*d:%*d:%*d: Char \"%[a-zA-Z ]\"",
right_letter);
        if (isupper(right_letter[0]))
           right_letter[0] = tolower(right_letter[0]);
/*         printf("%s %c", right_letter, right_letter[0]);*/
        percentage_so_far = 100.0;
        original_psf      = 100.0;
        characters_available = 27;
/*         printf("line %d: %s %d\n", line, results[line], in("Bet",
results[line]));       */
        for (line ++; in("Bet", results[line]); line ++)
       {
/*         printf("bet line %d: %s\n", line, results[line]); */

           sscanf(results[line], "At %d:%d:%d: Bet %f%% on \"%[ a-
z]\"",
                  &hours, &minutes, &seconds, &orig_percentage,
characters);
           percentage = orig_percentage * percentage_so_far /
original_psf;
           /* adjust percentage for new situation;
              if more percentage available, percentage will rise */
           characters_chosen = strlen(characters);
           breakeven = (float) characters_chosen /
characters_available ;
/*           printf("times %d %s %d %d %f\n", times, right_letter,
characters_chosen,
                                      characters_available, breakeven);
*/
           certain      = percentage_so_far *  MAX_PERCENT;
           very_sure    = percentage_so_far * (breakeven +
(MAX_PERCENT - breakeven) * .90);
           quite_sure   = percentage_so_far * (breakeven +
(MAX_PERCENT - breakeven) * .75);
```

```
        probably    = percentage_so_far * (breakeven +
(MAX_PERCENT - breakeven) * .60);
        likely      = percentage_so_far * (breakeven +
(MAX_PERCENT - breakeven) * .45);
        possibly    = percentage_so_far * (breakeven +
(MAX_PERCENT - breakeven) * .30);
        conceivably = percentage_so_far * (breakeven +
(MAX_PERCENT - breakeven) * .15);
        no_idea     = percentage_so_far *  breakeven;

/*printf("%f %f %f %f %f %f %f\n",  certain, very_sure,
quite_sure, probably,
                                    likely, possibly,
conceivably, no_idea);*/
        certain_distance     = abs(certain     - percentage);
        very_sure_distance   = abs(very_sure   - percentage);
        quite_sure_distance  = abs(quite_sure  - percentage);
        probably_distance    = abs(probably    - percentage);
        likely_distance      = abs(likely      - percentage);
        possibly_distance    = abs(possibly    - percentage);
        conceivably_distance = abs(conceivably - percentage);
        no_idea_distance     = abs(no_idea     - percentage);

/*  printf("%f %f %f %f %f %f %f %f\n",certain_distance,
very_sure_distance,
   quite_sure_distance,  probably_distance, likely_distance,
possibly_distance,
        conceivably_distance, no_idea_distance);*/

          if (very_sure_distance        > certain_distance)
            norm_percentage = certain;
          else if (quite_sure_distance  > very_sure_distance)
            norm_percentage = very_sure;
          else if (probably_distance    > quite_sure_distance)
            norm_percentage = quite_sure;
          else if (likely_distance      > probably_distance)
            norm_percentage = probably;
          else if (possibly_distance    > likely_distance)
            norm_percentage = likely;
          else if (conceivably_distance > possibly_distance)
            norm_percentage = possibly;
          else if (no_idea_distance     > conceivably_distance)
            norm_percentage = conceivably;
          else
            norm_percentage = no_idea;


        if (in(right_letter, characters))
        {
          characters_available  = characters_chosen;
          percentage_so_far = norm_percentage;
          original_psf      = orig_percentage;
        }
        else
        {
          characters_available -= characters_chosen;
          percentage_so_far -= norm_percentage;
          original_psf      -= orig_percentage;
        }
/*printf(" left %f\n", percentage_so_far);         */
        printf("At %d:%d:%d: Bet %f%% on \"%s\"\n",
              hours, minutes, seconds, norm_percentage,
characters);


      }
    printf("%s\n", results[line]);
    line ++;
    }
}


in(member, set)
char *member, *set;
{
  int i, in = 0;

  for (i = 0; i < strlen(set); i ++)
    in += (strncmp(member, &set[i], strlen(member)) == 0);
  return(in);
}
```

Info extractor for log files

```
#include <stdio.h>
#include <math.h>

char results[5000][100];
struct char_info
{
  char character;
  float probability,
        bits;
  int time_taken, attempts;
} info[1000];

  float total = 1;
main()
{
  int length, wins;

  length = read_results();
  wins = extract_info(length);
  print_info(wins);
}

read_results()
{
  int line;
  char ch = '\0', character;

  for (line = 0; ch != EOF; line ++)
    for (character = 0; ((ch = getchar()) != '\n') && (ch != EOF)
; character ++)
      if (ch == '\n')
        results[line][character] = '\0';
      else
        results[line][character] = ch;
  return (line);
}

extract_info(length)
int length;
{
  int line, last_win = 0,
  hours, minutes, seconds, now, last_time, wins = 0;
  char characters[100];
  float percentage;

  sscanf(results[0], "At %d:%d:%d", &hours, &minutes, &seconds);
  last_time = seconds + 60 * (minutes + 60 * hours);

  for (line = 0; line < length; line ++)
  {
    if (in("Win", results[line]))
    {
/*        printf("line %d: %s\n", line -1, results[line-1]); */

        sscanf(results[line - 1], "At %d:%d:%d: Bet %f%% on \"%[a-z
]\"",
                &hours, &minutes, &seconds, &percentage,
characters);
        info[wins].probability = percentage / 100;

        now = seconds + 60 * (minutes + 60 * hours);
        info[wins].time_taken = now - last_time;
        last_time = now;

        info[wins].attempts = line - last_win - 2;
        last_win = line;

        info[wins].character = characters[0];

        info[wins].bits = -log2(percentage / 100);

/*      printf("%f %% %c %d %d\n",
        percentage, info[wins].character, info[wins].time_taken,
info[wins].attempts); */

        total -= log2(percentage / 100);
        wins ++;
    }
  }
  return(wins);
}

print_info(wins)
int wins;
{
  int line;
  printf("Total number of bits = %g, compression = %g
bits/character\n\n",
        total, total / wins);

   printf("Char.\tNumber\tprobability\tbits\t\ttime\tattempts\n");
  for (line = 0; line < wins; line ++)
    printf("%c   \t%d\t%10.5f %%\t%9.5f\t%d\t%d\n",
    info[line].character,
    line,
    info[line].probability * 100,
    info[line].bits,
    info[line].time_taken,
    info[line].attempts);
```

```
}
in(member, set)
char *member, *set;
{
  int i, in = 0;

  for (i = 0; i < strlen(set); i ++)
    in = in | (strcmp(member, &set[i]) == 0);
  return(in);
}
```

condition for knocking files into shape

```
#include <stdio.h>
main()
{
char ch, lastch = '\0';
int space_last = 0;

  while ((ch = getchar() ) != EOF)
  {
    if (ch == '\n')
      ch = ' ';

    if (ch == ' ' && !space_last)
      {
        putchar(' ');
        space_last = 1;
      }


    if (ch >= 'a' && ch <= 'z')
    {
       putchar(ch);
       space_last = 0;
    }
    if (ch >= 'A' && ch <= 'Z')
    {
       putchar(ch - 'A' + 'a');
       space_last = 0;
    }
    if (ch == ':' && lastch >= '0' && lastch <= '9')
      putchar('\n');
    lastch = ch;
  }
}
```

to condition raw KJV text

```
#include <stdio.h>
main()
{
char ch, lastch = '\0';
int space_last = 0, newline = 1;

   while ((ch = getchar() ) != EOF)
   {
      if (ch == '.' && newline)
      {
         while (getchar() != '\n')
           ;
         ch = '\n';
      }

      if (ch == '\\' && newline)
      {
         putchar('\n');
         putchar('1');
         putchar('\t');
      }
      if (ch == '\n')
      {
        ch = ' ';
        newline = 1;
      }
      else
        newline = 0;

      if (ch == '_')
        ch = ' ';

      if (ch == '-' && lastch >= 'a' && lastch <= 'z')
        ch = ' ';

      if (ch >= '0' && ch <= '9')
      {
         putchar('\n');
         putchar(ch);
         while ((ch = getchar() ) != EOF && (ch >= '0' && ch <= '9'
|| ch == '-'))
           putchar(ch);
         putchar('\t');
      }

      if (ch == ' ' && !space_last)
        {
           putchar(' ');
           space_last = 1;
        }
      if (ch >= 'a' && ch <= 'z')
      {
         putchar(ch);
         space_last = 0;
      }
      if (ch >= 'A' && ch <= 'Z')
      {
         putchar(ch - 'A' + 'a');
         space_last = 0;
      }

      lastch = ch;
   }
}
```

to condition TEV files

```
#include <stdio.h>
main()
{
char ch, lastch = '\0', beforelastch = '\0';
int space_last = 0;

  while ((ch = getchar() ) != EOF)
  {
    if (ch == '\n')
      ch = ' ';
    if (ch == '-')
      ch = ' ';
    if (ch == ':' && lastch >= '0' && lastch <= '9')
    {
       putchar('\n');
       while ((ch = getchar()) >= '0' && ch <= '9')
         putchar(ch);
       putchar('\t');
    }

    if (ch == ' ' && !space_last)
      {
         putchar(' ');
         space_last = 1;
      }


    if (ch >= 'a' && ch <= 'z')
    {
       putchar(ch);
       space_last = 0;
    }
    if (ch >= 'A' && ch <= 'Z')
    {
       putchar(ch - 'A' + 'a');
       space_last = 0;
    }
    beforelastch = lastch;
    lastch = ch;
  }
}
```

splice.c - for splicing two parallel translations together.

```
#include <stdio.h>

FILE *one, *two;

main(argc,argv)
int argc;
char *argv[];
{

   char number1[10], number2[10], verse1[500], verse2[500];
   char out1[3000], out2[3000];
   int last1, last2, index1, index2;

   open_files(argc, argv);

   while (!feof(one) && !feof(two))
   {
      fscanf(one, "%s\t%[a-z ]\n", number1, verse1);
      fscanf(two, "%s\t%[a-z ]\n", number2, verse2);

     index1 = 0;
     index2 = 0;

      index1 = append(out1, index1, verse1);
      index2 = append(out2, index2, verse2);


      while ((last1 = last(number1)) != (last2 = last(number2))
          && !feof(one) && !feof(two))

        if (last1 > last2)
       {
          index2 = append(out2, index2, verse2);
           fscanf(two, "%s\t%[a-z ]\n", number2, verse2);
          last2 = last(number2);
       }
       else
       {
          index1 = append(out1, index1, verse1);
           fscanf(one, "%s\t%[a-z ]\n", number1, verse1);
          last1 = last(number1);
       }
        out1[index1] = '\0';
        out2[index2] = '\0';
         printf("$$%s\n$$%s\n", out1, out2);
   }
}

in(member, set)
char member, *set;
{
   int i, in = 0;

   for (i = 0; i < strlen(set); i ++)
     in = in | (member == set[i]);
   return(in);
}

last(range)
char range[];
{
   int first, last;
   if (in('-', range))
      sscanf(range, "%d-%d", &first, &last);
   else
   {
      sscanf(range, "%d", &last);
     first = last;
   }
   return last;
}

open_files(argc, argv)
int argc;
char *argv[];
{
   if (argc < 3)
   {
      fprintf(stderr, "usage: splice <version1> <version2>\n");
     exit(1);
   }

   if ((one = fopen(argv[1], "r")) == NULL)
   {
      fprintf(stderr, "Problems opening file %s\n", argv[1]);
     exit(1);
   }

   if ((two = fopen(argv[2], "r")) == NULL)
   {
      fprintf(stderr, "Problems opening file %s\n", argv[2]);
     exit(1);
   }
}


int append(out, index, verse)
int index;
char out[], verse[];
{
```

```
  int myindex;
  myindex = index;
  strcpy(&out[myindex], verse);
  myindex += strlen(verse);
  out[myindex] = ' ';
  myindex ++;
  return myindex;
}
```

Random parallel text generator using compressed file for random numbers!

```
#include <stdio.h>

struct pos
{
  char pos[5];
  char syn[16][200];
  int  no_syn;
};

struct word
{
  char word[20];
  struct pos posse[6];
  int no_pos;
} verse[200];


main(argc, argv)
int argc;
char *argv[];
{
  FILE *target, *fred;
  int i, eoverse, eolist, eopos, word_no, pos_no, syn_no, w, p, s;
  char words[20][100], source[20][100], pos[20], syn[20], ch;

  if ((fred =
fopen("/usr/users/honours/nevill/project/parallel/generate/genesis
.Z", "r")) == NULL)
    fprintf(stderr, "Help!");
  getchar(); /* get rid of newline at beginning */
  while (!feof(stdin))
  {
    eoverse = 0;
    i = 0;
    word_no = 0;
    eolist = 0;
    verse[word_no].no_pos = 0;
     scanf("%[a-z]", verse[word_no].word);
    pos_no = 0;
    if (getchar() != '\n')
      while (!eolist && !feof(stdin))
      {
        ch = getchar();
        if (ch == '~')
        {
          eolist = 1;
          getchar(); /* read the \n */
        }
        else
        {
          if (ch != '|')
            fprintf(stderr, "Parse error: missing bar\n");
           scanf("%[a-z]", verse[word_no].posse[pos_no].pos);
          if (getchar() != ':')
            fprintf(stderr, "Parse error: missing colon\n");
          eopos = 0;
          syn_no = 0;
          while (!eopos)
          {
            if (scanf("%[a-z' -]",
                    verse[word_no].posse[pos_no].syn[syn_no]) ==
0)
              eopos = 1;
            else
            {
              ch = getchar();
              if (ch != ',')
                fprintf(stderr, "Parse error: missing comma  \n");
            }
            syn_no ++;
          } /* while !eopos */
          verse[word_no].posse[pos_no].no_syn = syn_no;
          pos_no ++;
        }
      } /* while !eolist */
    verse[word_no].no_pos = pos_no;

  w = 0;

  if (verse[w].no_pos == 0)
    printf("%s ", verse[w].word);
  else
  {
    p = fgetc(fred) * verse[w].no_pos / 256;
    s = fgetc(fred) * verse[w].posse[p].no_syn / 256;
    printf("%s ", verse[w].posse[p].syn[s]);
  }

  }
}
```

Large Thesaurus ~~generator~~ decoder

```
#include <stdio.h>
#define BIT_MASK(b)  ((1<<b)-1)
#define BITS_IN_C    8            /* bits for each getchar */

static int bits_left;     /* bits available in current output byte
*/
static int c;             /* current output byte */


int bits_in = 0;

main()
{
   int i, end_of_block;
   unsigned int ch;
   char *pos[16];

   pos[0] =  "noun";
   pos[1] =  "one";
   pos[2] =  "adj.";
   pos[3] =  "verb";
   pos[4] =  "four";
   pos[5] =  "five";
   pos[6] =  "adv.";
   pos[7] =  "prep.";
   pos[8] =  "conj.";
   pos[9] =  "nine";
   pos[10] =  "ten";
   pos[11] = "eleven";
   pos[12] = "twelve";
   pos[13] = "thirteen";
   pos[14] = "fourteen";
   pos[15] = "fifteen";

   for (i = 0; i < 304 * 512; i ++)
     getchar();

   while (!feof(stdin))
   {
     end_of_block = 0;
     while (!end_of_block & !feof(stdin))
     {
       end_of_block = 0;
       ch = bitin(4);
       if (ch < 16)
          printf("\n%s:\t", pos[ch]);
       else
         break;

       if (((512 * 8) - (bits_in % (512 * 8))) <= 5)
       {
          if (bits_left)
            bitin(bits_left);
          break;
       }

       ch = 0;
       while (ch != 0x1f && !feof(stdin))
       {
          ch = bitin(5);
          if (ch == EOF)
            break;
          else if (ch == 0x1f)
          {
             if (bits_left)
               bitin(bits_left);
          }
          else if (ch == 0)
          {
             printf(", ");
             end_of_block = 1;
             if (((512 * 8) - (bits_in % (512 * 8))) <= 5)
             {
             if (bits_left)
                bitin(bits_left);
               break;
             }
          }
          else if (ch == 29)
            putchar(' ');
          else if (ch == 28)
            putchar('\'');
          else if (ch == 27)
            putchar('-');
          else
             putchar('a' + ch - 1);
       }
     }
   }
   putchar('\n');
}

/*
 * bitin(b)
 *
 * read in the value of the next b bits
 */

int bitin(b)
     int b;
```

```
{
  int n = 0;

#ifdef DEBUG
if (debug_io_flag)
  printf("bitin(%d) = ",b);
#endif

  bits_in += b;
  if (bits_left >= b) {
    bits_left -= b;
    n = (c>>bits_left) & BIT_MASK(b);
  }
  else {
    if (bits_left) {
      n = BIT_MASK(bits_left) & c;
      b -= bits_left;
      bits_left = 0;
    }
    while (b >= BITS_IN_C) {
      n = (n << BITS_IN_C) | getchar();
      b -= BITS_IN_C;
    }
    if (b) {
      c = getchar();
      bits_left = BITS_IN_C - b;
      n = (n<<b) | (c >> bits_left);
    }
  }

#ifdef DEBUG
if (debug_io_flag)
  printf("%d\n",n);
#endif

  return n;
}
```

Synonym deducer

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>
#define NOUN        0x1
#define VERB        0x2
#define ADJECTIVE   0x4
#define ADVERB          0x8
#define PRONOUN         0x10
#define CONJUNCTION     0x20
#define ARTICLE         0x40
#define PREPOSITION     0x80
#define PROPER_NOUN     0x100

#define BUCKETS         59999


struct syn
{
   struct word *word_ptr;
   short frequency;
   struct syn *next;
};

struct word
{
   char word[20];
   short pos;
   short own_frequency;
   short total_frequency;
   short inversion2;
   struct syn *synonym, *last_syn;
   struct word *next;
};

struct word *thesaurus[BUCKETS];

FILE *thes, *infile, *outnewt, *newt;

int total_chunks = 0, wc1, wc2, malloced = 0;

struct word *words1[500], *words2[500];
char version1[4000], version2[4000];

char *word();
struct word *find_or_insert();
struct word *so_far_find_or_insert();
main()
{
   int i, this_index, no_verses, total_words = 0, total_bytes = 0;
   char *this_word;
   struct syn *this_syn;
   struct word *this_ptr;
   struct rusage rusage[1];
   initialise();
   read_parts_of_speech();

   if ((thes = fopen("thes", "r")) != NULL)
      get_thesaurus_so_far(thes);

   newt = fopen("newt", "r");
   while (!feof(newt))
   {
      getrusage(RUSAGE_SELF, rusage);
      if (rusage->ru_majflt > 100)
      {
         fprintf(stderr, "\nPage faults have reached 100; passing the
baton...");
         results(fopen("thes", "w"));
         fprintf(stderr, "Printed results\n");
         outnewt = fopen("newt2", "w");
         fprintf(stderr, "Opened newt2\n");
            while (!feof(newt))
               putc(fgetc(newt), outnewt);
            fclose(outnewt);
         fprintf(stderr, "Renaming newt2 to newt\n");
            system("mv newt2 newt");
         fprintf(stderr, "Starting myself again\n");
         system("thesaurus &");
         fprintf(stderr, "Bye-bye!\n");
           exit(1);
      }

      if (fscanf(newt, "%[a-z ]\n%[a-z ]\n", version1, version2) ==
0)
         break;
      total_bytes += strlen(version1) + strlen(version2);
      fprintf(stderr, "%d\r", total_bytes);
      total_chunks ++;
      for (wc1 = 0; (this_word = word(wc1 + 1, version1)) != NULL;
wc1 ++)
      {
         words1[wc1] = find_or_insert(this_word);
         words1[wc1]->total_frequency ++;
/*         if (words1[wc1]->ignore)
           words1[wc1] = NULL;*/
      }

      for (wc2 = 0; (this_word = word(wc2 + 1, version2)) != NULL;
wc2 ++)
      {
```

```
      words2[wc2] = find_or_insert(this_word);
      words2[wc2]->inversion2 ++;
      words2[wc2]->total_frequency ++;
/*      if (words2[wc2]->ignore)
        words2[wc2] = NULL;*/
    }

    for (i = 0; i < wc1; i ++)
    {
      this_ptr = words1[i];
      if (this_ptr != NULL)
      {
        if (this_ptr->pos != 0)
          match(this_ptr);
        else
          matchall(this_ptr);
        total_words ++;
      }
    }
  }
  results(fopen("thes", "w"));


  }

results(outfile)
FILE *outfile;
{
int i, total_chunks = 0;
struct syn *this_syn;
struct word *word_ptr;

for (i = 0; i < BUCKETS; i ++)
  for (word_ptr = thesaurus[i];
       word_ptr != NULL;
       word_ptr = word_ptr->next)
    if (word_ptr->total_frequency > 0)
    {
      fprintf(outfile, "\n%s %d %d %d ", word_ptr->word,
             word_ptr->total_frequency, word_ptr->own_frequency,
             word_ptr->inversion2);

      for (this_syn = word_ptr->synonym;
           this_syn != NULL;
           this_syn = this_syn->next)
        if (this_syn->frequency > 1)
          fprintf(outfile, "%s %d ", this_syn->word_ptr->word,
           this_syn->frequency);
    }
  fprintf(outfile, "\n");
  fclose(outfile);
}

struct word *find_or_insert(this_word)
char *this_word;
{
  int i, len;
  unsigned long hash_value = 0;
  char *word_temp;
  struct word *word_ptr;

  word_temp = this_word;
  for (len = strlen(this_word); len != 0; len --)
  {
    hash_value *= 4;
    hash_value += *(word_temp ++);
   }
  hash_value %= BUCKETS;


  if (thesaurus[hash_value] != NULL)
  {
    for (word_ptr = thesaurus[hash_value];
         strcmp(word_ptr->word, this_word) != 0 && word_ptr->next
!= NULL;
         word_ptr = word_ptr->next)
      ;

    if (strcmp(word_ptr->word, this_word) == 0)
      return word_ptr;

    word_ptr->next = (struct word *) malloc(sizeof(struct word));
    word_ptr = word_ptr->next;
  }
  else
  {
    thesaurus[hash_value] = (struct word *) malloc(sizeof(struct
word));
    word_ptr = thesaurus[hash_value];
  }

  strcpy(word_ptr->word, this_word);
  word_ptr->pos = 0;
  word_ptr->own_frequency = 0;
  word_ptr->total_frequency = 0;
  word_ptr->inversion2 = 0;
/*  word_ptr->ignore = 0;*/
  word_ptr->synonym = NULL;
  word_ptr->last_syn = NULL;
  word_ptr->next = NULL;
```

```
  return(word_ptr);
}

match(word_ptr)
struct word *word_ptr;
{
  int i;

  for (i = 0; i < wc2; i ++)
    if (words2[i] != NULL)
      if ((word_ptr->pos & words2[i]->pos) || (words2[i]->pos ==
0))
        make_synonym(word_ptr, words2[i]);
}

matchall(word_ptr)
struct word *word_ptr;
{
  int i;

  for (i = 0; i < wc2; i ++)
    if (words2[i] != NULL)
      make_synonym(word_ptr, words2[i]);
}

make_synonym(this_ptr, other_ptr)
struct word *this_ptr, *other_ptr;
{
  struct syn *last_syn, *this_syn, *just_before;

  if (this_ptr == other_ptr)
  {
    this_ptr->own_frequency ++;
    return;
  }

  last_syn = this_ptr->last_syn;

  if (last_syn != NULL)
  {
    for (  this_syn = this_ptr->synonym;
           this_syn->next != NULL &&
           this_syn->word_ptr != other_ptr;
           this_syn = this_syn->next)
      just_before = this_syn;

    if (this_syn->word_ptr != other_ptr)
    {
      last_syn->next = (struct syn *)malloc(sizeof(struct syn));
      malloced ++;
      last_syn = last_syn->next;
      last_syn->word_ptr = other_ptr;
      last_syn->frequency = 1;
      this_ptr->last_syn = last_syn;
    }
    else
    {
      this_syn->frequency ++;
      if (this_syn != this_ptr->synonym )
      {
        just_before->next = this_syn->next;
        this_syn->next = this_ptr->synonym;
        this_ptr->synonym = this_syn;
      }
    }
  }
  else
  {
    this_ptr->synonym = (struct syn *)malloc(sizeof(struct syn));
    malloced ++;
    this_ptr->synonym->next = NULL;
    this_ptr->synonym->word_ptr = other_ptr;
    this_ptr->synonym->frequency = 1;
    this_ptr->last_syn = this_ptr->synonym;
  }

}

read_parts_of_speech()
{
  FILE *pos;
  char parts[20], *part, this_word[20];
  short pos_temp;
  int i = 0, j;
  struct word *word_ptr;

  if((pos = fopen("pos", "r")) == NULL)
  {
    fprintf(stderr, "Couldn't open pos (parts of speech) file\n");
    exit(1);
  }
  while (!feof(pos))
  {
    if (fscanf(pos, "%*d %s %[a-z ]\n", this_word, parts) < 2)
      fprintf(stderr, "%s untagged\n", this_word);

    pos_temp = 0;
    for (j = 1; (part = word(j, parts)) != NULL; j ++)
    {
```

```
      if ((strcmp(part, "n")   == 0) && strlen(part) == 1)
pos_temp |= NOUN;
      if ((strcmp(part, "v")   == 0) && strlen(part) == 1)
pos_temp |= VERB;
      if ((strcmp(part, "a")   == 0) && strlen(part) == 1)
pos_temp |= ADJECTIVE;
      if ((strcmp(part, "c")   == 0) && strlen(part) == 1)
pos_temp |= CONJUNCTION;
      if ((strcmp(part, "av")  == 0) && strlen(part) == 2)
pos_temp |= ADVERB;
      if ((strcmp(part, "pn")  == 0) && strlen(part) == 2)
pos_temp |= PRONOUN;
      if ((strcmp(part, "ar")  == 0) && strlen(part) == 2)
pos_temp |= ARTICLE;
      if ((strcmp(part, "pr")  == 0) && strlen(part) == 2)
pos_temp |= PREPOSITION;
      if ((strcmp(part, "prn") == 0) && strlen(part) == 3)
pos_temp |= PROPER_NOUN;
    }
    word_ptr = find_or_insert(this_word);
/*    printf("%s %x %s\n", this_word, pos_temp, parts);*/
    word_ptr->pos = pos_temp;
/*    if (i < 18)
      word_ptr->ignore = 1;*/
    i ++;
  }
}

char *word(n, string)
int n;
char string[];
{
  int i, j, space_last = 1;
  static char word[200];

  word[0] = '\0';

  for (i = 0; *string && n; string ++)
    if (string[i] == ' ')
        space_last = 1;
    else
    {
      if (space_last)
        n --;
      space_last = 0;
    }

  string --;
  if (n != 0)
  {
    return(NULL);
  }

  for (j = 0; *string && (*string != ' '); string ++, j ++)
    word[j] = *string;
  word[j] = '\0';

  return word;
}


initialise()
{
  int i;

  for (i = 0; i < BUCKETS; i ++)
    thesaurus[i] = NULL;
}

get_thesaurus_so_far(thes)
FILE *thes;
{
  int i, this_index, no_verses, total_words = 0, total_bytes = 0;
  char this_word[20], syn[20], this_line[1000], *char_ptr;
  short count;
  struct syn *this_syn;
  struct word *this_ptr, *other_ptr;
  short total_frequency, own_frequency, inversion2;
  int no_words = 0, threshold;

  while (!feof(thes))
  {
    fscanf(thes, "\n%[a-z]", this_word);
    this_ptr = so_far_find_or_insert(this_word);
    fscanf(thes, " %hd %hd %hd", &this_ptr->total_frequency,
          &this_ptr->own_frequency, &this_ptr->inversion2);
    while (fgetc(thes) != '\n' && !feof(thes))
    {
      if (fscanf(thes, "%[a-z] %hd", syn, &count) == 0)
        break;
      other_ptr = so_far_find_or_insert(syn);
      so_far_make_synonym(this_ptr, other_ptr, count);
    }
    no_words ++;
    fprintf(stderr, "%d\r", no_words);
  }
  fprintf(stderr, "\nProcessing file...\n");
}
```

```
struct word *so_far_find_or_insert(this_word)
char *this_word;
{
  int i, len;
  unsigned long hash_value = 0;
  char *word_temp;
  struct word *word_ptr;

  word_temp = this_word;
  for (len = strlen(this_word); len != 0; len --)
  {
    hash_value *= 4;
    hash_value += *(word_temp ++);
   }
  hash_value %= BUCKETS;


  if (thesaurus[hash_value] != NULL)
  {
    for (word_ptr = thesaurus[hash_value];
         strcmp(word_ptr->word, this_word) != 0 && word_ptr->next
!= NULL;
         word_ptr = word_ptr->next)
      ;

    if (strcmp(word_ptr->word, this_word) == 0)
      return word_ptr;

    word_ptr->next = (struct word *) malloc(sizeof(struct word));
    word_ptr = word_ptr->next;
  }
  else
  {
    thesaurus[hash_value] = (struct word *) malloc(sizeof(struct
word));
    word_ptr = thesaurus[hash_value];
  }

  strcpy(word_ptr->word, this_word);
  word_ptr->pos = 0;
  word_ptr->own_frequency = 0;
  word_ptr->total_frequency = 0;
  word_ptr->inversion2 = 0;
/*  word_ptr->ignore = 0;*/
  word_ptr->synonym = NULL;
  word_ptr->last_syn = NULL;
  word_ptr->next = NULL;

  return(word_ptr);
}


so_far_make_synonym(this_ptr, other_ptr, count)
struct word *this_ptr, *other_ptr;
short count;
{
  struct syn *last_syn, *this_syn, *just_before;

  last_syn = this_ptr->last_syn;

  if (last_syn != NULL)
  {
    for (  this_syn = this_ptr->synonym;
           this_syn->next != NULL;
           this_syn = this_syn->next)
      ;
    last_syn->next = (struct syn *)malloc(sizeof(struct syn));
    malloced ++;
    last_syn = last_syn->next;
    last_syn->word_ptr = other_ptr;
    last_syn->frequency = count;
    this_ptr->last_syn = last_syn;
  }
  else
  {
    this_ptr->synonym = (struct syn *)malloc(sizeof(struct syn));
    malloced ++;
    this_ptr->synonym->next = NULL;
    this_ptr->synonym->word_ptr = other_ptr;
    this_ptr->synonym->frequency = count;
    this_ptr->last_syn = this_ptr->synonym;
  }

}


clean_up()
{
  int i, total_chunks = 0, freed = 0;
  struct syn *this_syn, *just_before;
  struct word *word_ptr;

  fprintf(stderr, "Cleaning up: ");
  for (i = 0; i < BUCKETS; i ++)
    for (word_ptr = thesaurus[i];
         word_ptr != NULL;
         word_ptr = word_ptr->next)
      for (this_syn = word_ptr->synonym;
           this_syn != NULL;
```

```
            this_syn = this_syn->next)
        {
          if (this_syn->frequency == 1)
          {
           if (this_syn == word_ptr->synonym)
            {
              word_ptr->synonym = this_syn->next;
              free(this_syn);
              malloced --;
            }
            else
            {
              just_before->next = this_syn->next;
              free(this_syn);
              malloced --;
            }
          }
          else
            just_before = this_syn;

        }
  fprintf(stderr, "%d freed\n", 300000 - malloced);
}



in(member, set)
char *member, *set;
{
  int i, in = 0;

  for (i = 0; *set != '\0'; set ++)
    in = in | (strcmp(member, set) == 0);
  return(in);
}
```