Tallinna Tehnikaülikool
Informaatikainstituut

Täheühendite eraldamine

iseseisev töö #5 õppeaines
Eksperimentaalsed sõresüsteemid
IDN5590

juhendaja : Rein Kuusik

üliõpilane : Erki Suurjaak
e-post : erki@lap.ttu.ee
matrikkel : 970772
õpperühm : LAP62

Tallinn 2000

Sisukord

Sisukord 2
1. Meetod 3
1.1. Tutvustus 3
1.2. Algoritm 3
2. Programm 4
2.1. Tutvustus 4
2.2. Programmi kasutajajuhend 5
2.3. Programmi väljund 5
2.4. Programmi listing 7

1. Meetod

1.1. Tutvustus

Meetod "täheühendite eraldamine" analüüsib sisendsõnade hulgas sisalduvate täheühendite esinemissagedust. Käesolev projekt tegeleb meetodi ühe realisatsiooniga, analüüsides täheühendite esinemissagedust sõna alguse suhtes. Näiteks, kui sisendsõnadeks on "meri mets mersu", siis analüüsi tulemusena leitakse, et täheühendit "m" esineb 3 korda, täheühendit "me" 3 korda, täheühendit "mer" 2 korda, täheühendit "meri" 1 kord, täheühendit "mers" 1 kord, täheühendit "mersu" 1 kord, täheühendit "met" 1 kord ja täheühendit "mets" 1 kord.

1.2. Algoritm

Realisatsiooni algoritm on järgmine :

1 loetakse sisendsõnade hulka, kuni jõutakse sümbolini, mis ei ole sõnade eraldaja
2 lisatakse käesolev sümbol vastavasse kohta täheühendite esinemise puus
3 loetakse sisendsõnade hulgast; kui loetud sümbol ei ole sõnade eraldaja, siis minnakse punkti 2
4 kui sisendsõnade hulk ei ole tühi, siis minnakse punkti 1

2. Programm

2.1. Tutvustus

Programm on kirjutatud programmeerimiskeeles C++. Programmi levitatav versioon on kompileeritud programmeerimiskeskkonna DJGPP v2.03 koosluses oleva kompilaatoriga gcc v2.95 MS-DOS keskkonnas. Programm töötab MS-DOS keskkonnas.

Programmi kiirus ja analüüsitava faili võimalik suurus ei ole üheselt määratud - see on otseselt sõltuvuses kasutatava arvuti kiirusest ja mälumahust; samuti ka sisendfaili enda struktuurist. Palju redundantsi sisaldava sisendfaili analüüs võtab vähem aega ning mälu.

Programmis on võimalik valida kahe analüüsi variandi vahel - tavaanalüüsi ja tervikanalüüsi vahel. Tavaanalüüsi puhul on rõhutud sisendfaili võimalikule suurusele - algoritm on järgmine

1 i = 0x21
2 loetakse sisendsõnade hulka, kuni jõutakse sümbolini, mis ei ole sõnade eraldaja; sümbol peab võrduma i-ga; kui jõutakse sisendsõnade hulga lõppu, siis minnakse punkti 6
3 lisatakse käesolev sümbol vastavasse kohta täheühendite esinemise puus
4 loetakse sisendsõnade hulgast; kui loetud sümbol ei ole sõnade eraldaja, siis minnakse punkti 3
5 kui sisendsõnade hulk ei ole tühi, siis minnakse punkti 2
6 kirjutatakse analüüsi tulemused faili
7 i = i + 1; kui i < 0xFE, siis minnakse punkti 2

Lahtiseletatult - sisendfailis otsitakse korraga ainult mingi kindla sümboliga algavaid sõnu. Sisendfaili lõppu jõudes kirjutatakse tehtud analüüs faili ja võetakse uus algsümbol. Faili töötatakse läbi niimitu korda, kuipalju on failis erinevaid sõnaalguse sümboleid.

Tervikanalüüsi puhul loetakse kogu fail korraga mällu ja analüüsitakse kõiki leiduvaid sõnu. Tervikanalüüsi puhul võtab programm tavaliselt rohkem mälu, aga on tõenäoliselt kiirem.

Näide programmi kiiruse kohta - arvuti peal, kus on *Celeron* 433 MHz protsessor ja 128 MB põhimälu, võttis ~9.5 MB faili tavaanalüüs ligikaudu 5 minutit, tervikanalüüs 0.5 minutit.

Sisendfail peab olema tavalise teksti kujul. Programm peab sõnu eraldavateks sümboliteks interpunktsioonimärke (punkti, koma, koolonit, semikoolonit, küsimärki,

hüüumärki) ja kõiki sümboleid, mille ASCII kood on väiksem kui 0x21 või suurem kui 0xFE.

2.2. Programmi kasutajajuhend

Programmile antakse parameetritega ette sisendfaili nimi, väljundfaili nimi ning parameetrid "/o" ning "/a" (ilma jutumärkideta; parameetrid on tõstutundetud). Nendest kohustuslik on ainult sisendfaili nimi - kui väljundfaili nime pole antud, siis suunatakse analüüsi tulemused ekraanile (täpsemalt voogu *stdout*).

Parameetriga "/o" käivitatult annab programm ülevaadet töö käigust - millega parasjagu tegeletakse. Ülevaade suunatakse ekraanile (täpsemalt voogu *stderr*). NB! Seda parameetrit ei ole soovitatav kasutada, kui väljundfaili nime pole antud ja ei ole rakendatud mingeid meetmeid voogude *stdout* ja *stderr* eraldamiseks (sellisel juhul analüüsi tulemused ning töö ülevaade segunevad).

Parameetriga "/a" käivitatult teeb programm tervikanalüüsi; vaikimisi tehakse tavaanalüüs.

2.3. Programmi väljund

Programmi tavaväljundi näide :

```
Analysis started at Wed Mar 15 06:56:29 2000
Input file name : m

3           m
3           me
2           mer
1           meri
1           mers
1           mersu
1           met
1           mets

Word count : 3
Different combinations : 8
All combinations : 13
Analysis finished at Wed Mar 15 06:56:29 2000
```

Sõnaseletusi :

Word count	Failis sisalduvate sõnade koguarv
Different combinations	Failis sisalduvate erinevate täheühendite koguarv
All combinations	Failis sisalduvate täheühendite koguarv

2.4. Programmi listing

```
/*
Author       : Erki Suurjaak
Created      : 20.02.2000
Last changed : 15.03.2000

The program analyzes the input file and determines the occurrence frequency
of all symbol combinations relative to the beginning of the word.

*/

#include <iostream>
#include <fstream>
#include <string.h>
#include <new.h>
#include <stdio.h>
#include <time.h>
#include <io.h>



// --------------------------- Global constants ---------------------------


const char* const ErrorMessageNoMoreMemory = "\nMemory allocation failed : not enough memory.\n";
const char* const ErrorMessageInputFileTooLarge = "\nThe specified file is too large. Maximum allowed file size is
5 million bytes.\n";
const char* const ErrorMessageInputFileError = "\nThere was an error reading from the specified input file.\n";
const char* const ErrorMessageOutputFileError = "\nThere was an error writing to the specified output file.\n";
const char* const EmptySpace = "  "; // An empty space, used in TSymbolTree::WriteTree
const char* const OverviewParameter = "/o";
const char* const AllAtOnceParameter = "/a";
const char* const HelpMessage =
  "\n"
  "This program analyzes the input file, determining the occurrence frequency\n"
  "of all symbol combinations relative to the beginning of the word they occur in.\n"
  "---\n"
  "Input file specification - space, dot, comma, colon, semicolon, exclamation\n"
  "mark, question mark and all symbols with byte code lower than 0x20 or higher\n"
  "than 0xFE are considered to be delimiting characters.\n"
  "---\n"
  "Output can be written to screen or to a file.\n"
  "---\n"
  "Use the '/o' switch to see an overview of the process.\n"
  "Use the '/a' switch for all-at-once analysis. It is usually faster, but more\n"
  "memory-hungry.\n"
  "---\n"
  "Program usage syntax : program.exe inputfile [outputfile] [switches]\n"
  "NB! The inputfile name must always come as the first and outputfile name,\n"
  "if any, as the second parameter (none = output to screen).\n";

const unsigned char MaxSymbols  = 222;  // Maximum number of allowed characters
const unsigned char FirstSymbol = 33;   // First allowed symbol
const unsigned char LastSymbol  = 254;  // Last allowed symbol


// -------------------------- /Global constants ---------------------------



// --------------------------- Global functions ---------------------------


// Displays Message and exits the program with errorlevel 255
```

```
inline void FatalError (const char* const Message)
{
 if (cerr) cerr << Message;
 exit (255);
 return;
}



inline void NoMoreMemory ()             // Function to be called when
{                                       // memory allocation fails.
  FatalError (ErrorMessageNoMoreMemory);
  return;
}


// Checks if the Symbol is a delimiting character
inline bool IsDelimiter (const unsigned char Symbol)
{
 // Punctuation marks are delimiting
 if (Symbol == '.' || Symbol == ',' || Symbol == ':'
     || Symbol == ';' || Symbol == '!' || Symbol == '?') return true;
 // Everything else between 21 and 254 is not delimiting
 if (Symbol >= FirstSymbol && Symbol <= LastSymbol) return false;

 // Everything remaining is delimiting
 return true;
}


// -------------------------- /Global functions --------------------------




// -------------------------- class TSymbolTree --------------------------

/*
  Class TSymbolTree is used to hold the symbol combination trees. Each
  tree node references a symbol in a fixed place. Each node has an element
  count - i.e. how many combinations it holds; and at most MaxSymbols
  subtrees - those mark the continuations of the symbol combinations.
  The amount of elements a node has is never less than the amount of
  subtrees it has. For example, to hold the words "meri mets mersu" the tree
  would look something like this:

    m
    3\
      e
     /3\
    r   t
   /2\  1\
  i   s   s
  1   1\  1
        u
        1

*/


class TSymbolTree
{
 public  :
         unsigned long Elements_;      // Element count in the current node
         TSymbolTree** SubTree_;       // Substring arrays

         // When handling long words, it is more efficient to construct a
         // string that would hold all of the word right at the beginning.
         // If the word is N characters long, the amount of memory required
         // is N + 1 bytes. If constructing a new string for each node,
         // the amount of memory required is
```

```
        // for (i = 2, Mem = 0; i <= N + 1; i++) Mem += i;
        // which can be very costly - when N == 10000, Mem == ~50 megabytes.
        static char* StringToWrite_;  // Used to hold the current string to write to output file
        static unsigned long StringToWriteLength_; // The current length of StringToWrite

 // Methods
        TSymbolTree ()
         {
          Elements_ = 0;
          SubTree_ = new TSymbolTree*[MaxSymbols];
          // Better safe than sorry
          for (int i = 0; i < MaxSymbols; i++) SubTree_[i] = NULL;
          return;
         }

        ~TSymbolTree ()
         {
          for (int i = 0; i < MaxSymbols; i++) delete SubTree_[i];
          delete [] *SubTree_;
          return;
         }


        inline void WriteTree (ostream*& out)
         {
          int i; // Just a variable for iterations
          char* TempChar = new char[11]; // Used for formatting
          for (i = sprintf (TempChar, "%u", Elements_); i < 10; i++)
              TempChar[i] = ' ';
          // The result is a string that has Elements_ in front and padding spaces up to the 10. symbol
          TempChar[10] = NULL;
          *out << TempChar << EmptySpace << StringToWrite_ << endl;
          delete TempChar;
          for (i = 0; i < MaxSymbols; i++)
           {
            if (SubTree_[i])
             {
              StringToWrite_[StringToWriteLength_] = char (i + FirstSymbol);
              StringToWriteLength_++;
              SubTree_[i]->WriteTree (out);
              StringToWriteLength_--;
              StringToWrite_[StringToWriteLength_] = NULL;
             }
           }
         }

};


// Initializing static data members
char* TSymbolTree::StringToWrite_ = NULL;
unsigned long TSymbolTree::StringToWriteLength_ = NULL;


// ------------------------ /class TSymbolTree -------------------------




int main (int argc, char** argv)
{
 // Output will be directed to screen if no output filename given
 bool OutputToScreen = false,
      OverviewDisplayed = false, // Whether process overview is displayed
      DoingAllAtOnce = false; // Whether fast, but memory-costly algorhithm is used
 unsigned long i; // Just a variable for iterations

 switch (argc)
 {
  case 1 : if (cout)
                  cout << HelpMessage;
```

```
          exit (0);
          break;
 case 2 : OutputToScreen = true; // Only inputfile given; output will be directed to screen
          break;
 default : for (i = 2; i < argc; i++)
            {
             if (!stricmp (argv[i], OverviewParameter))
                OverviewDisplayed = true;
             else if (!stricmp (argv[i], AllAtOnceParameter))
                DoingAllAtOnce = true;
             // Outputfile name must be given as the second parameter
             if ((OverviewDisplayed || DoingAllAtOnce) && i == 2)
                OutputToScreen = true;
            }
           break;
}

// Setting up memory allocation failure handling
set_new_handler (NoMoreMemory);

// The root of the symbol combinations tree.
TSymbolTree* Root = new TSymbolTree;
TSymbolTree* CurrentNode; // Used as a temporary pointer

unsigned long
     DifferentCombinations = NULL, // The combination count
     Words = NULL, // The overall word count (a word is something that has delimiters before and after)
     Combinations = NULL, // The overall combination count (including repeating combinations)
     LongestWordLength, // Holds the length of the longest word of the current starting character.
     CurrentWordLength; // Holds the length of the current, processing word


ofstream OutputFile; // The output file
ostream* out;        // Will be used for output; references either cout or OutputFile
if (OutputToScreen)
 {
  out = &cout; // Pointing out to cout. out is used for output.
 }
else
 {
  OutputFile.open (argv[2]);
  // If the file cannot be opened, the program cannot continue
  if (!OutputFile) FatalError (ErrorMessageOutputFileError);
  out = &OutputFile;
 }

time_t now;
time (&now);
*out << "Analysis started at " << asctime (localtime (&now));
*out << "Input file name : " << argv[1] << endl;
if (!OutputToScreen) *out << "Output file name : " << argv[2] << endl;
*out << endl;


if (DoingAllAtOnce) // If analyzing all the symbols at once
 {
  FILE *InputFile;
  unsigned long FileLength;
  unsigned char* File;

  if ((InputFile = fopen (argv[1], "rb")) != NULL)
   {
   FileLength = filelength (fileno(InputFile));

   File = new unsigned char[FileLength + 2];
   File[0] = NULL;               // First and last characters set to a delimiting
   File [FileLength + 1] = NULL; // character to avoid creating exceptions

   fread (File + 1, FileLength, 1, InputFile);
   if (OverviewDisplayed)
      cerr << "Inputfile has been read.\n";
```

```
    }
   else
    {
     FatalError (ErrorMessageInputFileError);
    };
   fclose (InputFile);

   Root = new TSymbolTree;

   i = 0;
   LongestWordLength = 0;

   // Parse File and build the tree
   while (i <= FileLength)
    {

     // Advance to the start of the word.
     while (IsDelimiter(File[i]) && i <= FileLength) i++;

     if (i > FileLength) break; // If there were no more words, the while block must be exited to avoid
incrementing Words

     CurrentWordLength = 0; // Holds the length of the current word. This and LongestWordLength are then compared
to see which is longer.

     CurrentNode = Root; // Move the pointer to the root of the tree
     Words++; // Keeping track of the overall word count
     while (!IsDelimiter(File[i]) && i <= FileLength)
      {
       CurrentWordLength++;
       // If the element is yet empty
       unsigned char ElementPosition = File[i] - FirstSymbol;
       if (!CurrentNode->SubTree_[ElementPosition])
        {
         CurrentNode->SubTree_[ElementPosition] = new TSymbolTree;
         DifferentCombinations++; // Keeping count of different combinations
        }
       Combinations++; // Keeping track of the overall combination count
       // Keeping count of the elements
       CurrentNode->SubTree_[ElementPosition]->Elements_++;
       // Move the pointer to the new position
       CurrentNode = CurrentNode->SubTree_[ElementPosition];
       i++; // Move the counter to the next symbol
      };
     if (CurrentWordLength > LongestWordLength)
        LongestWordLength = CurrentWordLength;
    }

   if (OverviewDisplayed)
      cerr << "Inputfile has been processed.\n";

   delete File;  // This is no longer needed


   for (i = 0; i < MaxSymbols; i++)
     if (Root->SubTree_[i])
        {
         Root->StringToWrite_ = new char[LongestWordLength + 1];
         Root->StringToWriteLength_ = 1;
         for (int j = 1; j <= LongestWordLength; j++)
             Root->StringToWrite_[j] = NULL;
         Root->StringToWrite_[0] = i + FirstSymbol;
         Root->SubTree_[i]->WriteTree (out);
         delete Root->StringToWrite_;
         delete Root->SubTree_[i];
         Root->SubTree_[i] = NULL;
        }

   if (OverviewDisplayed)
      cerr << "Outputfile has been written.\n";
```

```
  }
 else // If performing a symbol-by-symbol analysis
  {
   ifstream InputFile;       // The input file

   InputFile.open (argv[1]);
   // If the file cannot be opened, the program cannot continue
   if (!InputFile) FatalError (ErrorMessageInputFileError);

   if (OverviewDisplayed)
      cerr << "Parsing input file for the first time." << endl;

   unsigned char CurrentChar = NULL, // A variable for reading from file
                 ElementPosition; // To mark the current position in a SubTree_

   // First, the input file is parsed to determine all starting characters.
   // Later, only those subtrees are constructed.
   while (!InputFile.eof())
    {
     // Advance to the start of the word. EOF must be checked.
     while (IsDelimiter(CurrentChar) && !InputFile.eof())
           InputFile.get (CurrentChar);
     // If the starting symbol has not occurred yet, construct the appropriate branch
     if (!Root->SubTree_[CurrentChar - FirstSymbol])
      {
       Root->SubTree_[CurrentChar - FirstSymbol] = new TSymbolTree;
       DifferentCombinations++;
      }
     // Advance to the end of the word. EOF must be checked.
     while (!IsDelimiter(CurrentChar) && !InputFile.eof())
           InputFile.get (CurrentChar);
    }

   InputFile.close();

   CurrentChar = NULL;

   // Construct all necessary subtrees, write them to the output file, and
   // delete them.
   for (i = 0; i < MaxSymbols; i++)
    if (Root->SubTree_[i])
     {
      InputFile.open (argv[1]);
      if (OverviewDisplayed)
         cerr << "Now analyzing words starting with " << (unsigned char)(i + FirstSymbol) << endl;
      CurrentChar = NULL;
      LongestWordLength = NULL;
      while (!InputFile.eof ())
       {
        // Advance to the start of the word. EOF must be checked.
        while (IsDelimiter(CurrentChar) && !InputFile.eof())
              InputFile.get (CurrentChar);

        if (InputFile.eof()) break; // If there were no more words, the while block must be exited to avoid
incrementing Words

        CurrentWordLength = NULL; // Holds the length of the current word. This and LongestWordLength are then
compared to see which is longer.

        if (CurrentChar == i + FirstSymbol) // Only those words are analyzed that start with the current starting
symbol, i + FirstSymbol
         {
          CurrentNode = Root; // Move the pointer to the root of the tree
          Words++; // Keeping track of the overall word count
          while (!IsDelimiter(CurrentChar) && !InputFile.eof())
           {
            CurrentWordLength++;
            ElementPosition = CurrentChar - FirstSymbol;
```

```
            if (!CurrentNode->SubTree_[ElementPosition])  // If the element is yet empty
               {
                CurrentNode->SubTree_[ElementPosition] = new TSymbolTree;
                DifferentCombinations++; // Keeping count of different combinations
               }
            Combinations++; // Keeping track of the overall combination count
            CurrentNode->SubTree_[ElementPosition]->Elements_++; // Keeping count of the elements
            CurrentNode = CurrentNode->SubTree_[ElementPosition]; // Move the pointer to the new position
            InputFile.get (CurrentChar);  // Read the next character
           };
          if (CurrentWordLength > LongestWordLength)
             LongestWordLength = CurrentWordLength;
        }
       else while (!IsDelimiter(CurrentChar) && !InputFile.eof())
             InputFile.get (CurrentChar); // The symbol was not to be analyzed now
      }
     InputFile.close ();

     Root->StringToWrite_ = new char[LongestWordLength + 1];
     Root->StringToWriteLength_ = 1;
     for (int j = 1; j <= LongestWordLength; j++)
         Root->StringToWrite_[j] = NULL;
     Root->StringToWrite_[0] = i + FirstSymbol;
     Root->SubTree_[i]->WriteTree (out);
     delete Root->StringToWrite_;
     delete Root->SubTree_[i];
     Root->SubTree_[i] = NULL;
    }

  }

 delete Root;

 *out << endl;
 *out << "Word count : " << Words << endl;
 *out << "Different combinations : " << DifferentCombinations << endl;
 *out << "All combinations : " << Combinations << endl;
 time (&now);
 *out << "Analysis finished at " << asctime (localtime (&now));
 if (!OutputToScreen) OutputFile.close ();

 if (OverviewDisplayed) cerr << "Done." << endl;
 return 0;

}
```