NAME

ngram - eine Software zur Wortvorhersage.

SYNOPSIS

ngram [modus] [argument 2] [argument 3] ([argument 4])

DESCRIPTION

Diese Programm erstellt aus einem beliebigen Text Ngramme der Grösse N=3, sortiert diese und speichert sie in einer Datei ab. Anhand dieser Datei kann das Programm in einem zweiten Schritt sagen welche Wörter am wahrscheinlichsten auf einen Text folgen werden. Das Programm verfügt zudem über einen Testmodus bei dem die Häufigkeit berechnet wird, mit der in einem belibigen Text die korrekten Wörter vorgeschlagen worden wären und wie viele Tastenanschläge mit dieser Wortvorhersage hätten eingespart werden können.

OPTIONS

[modus] kann folgende Werte haben, die Argumente 2 bis 4 haben je nach Modus eine andere Funktion.

-"create":
Erstellt aus einem Text eine Ngram-Datenbank und speichert diese in einem Textfile.
[Argument 2] Pfad des Textes aus dem die Ngramme erstellt werden
[Argument 3] Ort an dem die erstellte Datenbank gespeichert wird

-"predict":
Sagt aufgrund einer durch "create" erstellten Datenbank die wahrscheinlichsten Wörter voraus, die auf einen Text folgen.
[Argument 2] Text für den die wahrscheinlichsten Wörter vorhergesagt werden sollen
[Argument 3] Pfad der Datenbank
[Argument 4] Pfad der Bufferdatei, in die der Text kopiert wird

-"test":
Zählt die Anzahl der Wörter, die in einem Text korrekt vorgeschlagen worden wären.
[Argument 2] Pfad der Datenbank
[Argument 3] Pfad des Textes

-"advtest":
Zählt die Anzahl der Tastendrücke die für das Eingeben eines Textes mit und ohne Wortvorhersagemethode nötig wären. Nicht berücksichtigt werden dabei Trennzeichen wie Leerschläge, Kommata, Punkte, Ausrufe- und Fragezeichen.
[Argument 2] Pfad der Datenbank
[Argument 3] Pfad des Textes
[Argument 4] Anzahl der maximal vorgeschlagenen Wörter

-"merge":
Fügt 2 Datenbank-Dateien in eine Datenbank-Datei zusammen.
[Argument 2] Pfad der einen Datenbank-Datei
[Argument 3] Pfad der anderen Datenbank-Datei
[Argument 4] Ort an dem die neu erstellte Datenbank-Datei gespeichert wird

FILES

Programm/ngram
Das hier beschriebene und kompilierte Programm.

Programm/text1.txt, ./text2.txt ./text3.txt ./text4.txt ./text5.txt
Verschiedene Texte um das Programm zu testen. Texte 2, 3 und 4 sind verschiedene englische Komplettlösungen zum selben Computerspiel (Zelda: Twilight Princess). Sie stammen von der Internetseite www.gamefaqs.com und sind unter Copyright der betreffenden Autoren. Da diese Texte im Internet frei verfügbar sind, habe ich es mir erlaubt diese als Beispiele zusammen mit diesem Programm abzugeben. Text 1 ist ein kurzer, Text 5 ein noch kürzerer Ausschnitt aus Text 4.

Programm/data1.txt, ./data2.txt ./data4.txt ./data24.txt
Das sind die aus den entsprechenden Texten erstellten Datenbanken. "data24.txt" ist die aus "data2.txt" und "data4.txt" zusammengeführte Datenbank. Da diese Datenbanken mit einer älteren nicht ganz fehlerfreien Version des Programmes erstellt wurden, fehlen vereinzelte Ngramme.

Projekt und Quellcode/main.cpp

Der gesamte Sourcecode des Programmes

Projekt und Quellcode/Makefile
Das Makefile um das Programm zu kompilieren. Seit dem Wechsel zu OSX 10.6 funktioniert dieses bei mir nicht mehr. Ich kann daher keine Garantie dafür übernehmen. Ich habe den Code mit XCode 3.0 kompiliert.

Projekt und Quellcode/ngram.xcodeproj
Das XCode 3.0 Projektfile mit dem das Programm kompiliert wurde.

Projekt und Quellcode/fstream.h
Headerfile für das Lesen und Schreiben von externen Files.

DIAGNOSTICS

Das Programm überprüft nicht ob die Argumente oder Files den Konventionen entsprechen. Falls ein Pfad zu einer Eingangs-Datei falsch ist, ist das Resultat identisch, wie wenn die Datei leer ist. Falls eine Datenbank-Datei nachträglich verändert worden ist und nicht mehr den Konventionen entspricht, wird dies zu unbekannten Abstürzen des Programms führen. Ebenso wenn ein für den entsprechenden Modus notwendiges Argument fehlt oder nicht den Konventionen entspricht. Durch falsch positionierte Argumente können auch Files gelöscht werden.

Beim Erstellen der Datenbank (Modus "create") kann das Sortieren der Ngramme je nach Länge des Textes sehr viel Zeit in Anspruch nehmen. Für einen Text mit 100'000 Wörter kann das Sortieren über eine halbe Stunde dauern. Schneller geht es wenn kleinere Texte verwendet werden und die Resultierenden Datenbank-Dateien im Modus "merge" zusammengefügt werden.

Die Argumente sollten alle in Anführungszeichen Eingegeben werden: z.B. "datei.txt".

EXAMPLES

Erstelle eine Datenbank anhand des Files "text1.txt" und speichere es als "data1.txt" ab.

```
>> ./ngram "create" "text1.txt" "data1.txt"
```

Füge die Datenbanken "data2.txt" und "data4.txt" zu einer grossen Datenbank zusammen und speichere diese als "data24.txt".

```
>> ./ngram "merge" "data2.txt" "data4.txt" "data24.txt"
```

Gib anhand der Datenbank "data24.txt" Wortvorschläge die auf den Text "I want to use " folgen könnten.

```
>> ./ngram "predict" "I want to use " "data24.txt" "bufferfile"
```

Gib anhand der Datenbank "data24.txt" Wortvorschläge die auf den Text "I want to use " folgen könnten und mit "t" beginnen.

```
>> ./ngram "predict" "I want to use t" "data24.txt" "bufferfile"
```

Teste wie häufig im Text "text5.txt" das korrekte Wort im passenden Ngram der Datenbank "data24.txt" vorhanden gewesen wäre.

```
>> ./ngram "test" "data24.txt" "text5.txt"
```

Teste wie viele Tastendrücke beim eingeben vom Text "text5.txt" gespart würden, wenn stets die 10 wahrscheinlichsten Wörter anhand der Datenbank "data24.txt" vorgeschlagen würden. (Notiz: Es braucht ebenfalls einen Tastendruck um eines der vorgeschlagenen Wörter auszuwählen. Leer- und Satzzeichen werden nicht gezählt.)

```
>> ./ngram "advtest" "data24.txt" "text5.txt" "10"
```

NOTES

Der gesamte Source-Code zum Programm "ngram" befindet sich in "main.cpp". Das Programm

ist in C++ geschrieben. Als Grundbaustein (Makefile, Projekt etc.) wurde eine Vorlage aus einer anderen Vorlesung verwendet. Das Makefile läuft bei mir seit dem Wechsel zu OSX 10.6 nicht mehr. Deshalb konnte ich nicht testen ob es sich mit diesem kompilieren lässt und kann keine Garantie übernehmen. Ich habe das Programm mit XCode 3.0 unter OSX 10.6 kompiliert. Das kompilierte Programm wird "exercise" getauft und muss von Hand in "ngram" umbenannt werden.

Alle Klassen, Procedures und der Ablauf befindet sich in main.cpp. Der Programmcode selbst ist nur sehr rudimentär doumentiert. Welche Programmteile das Programm beinhaltet, wie diese funktionieren, was sie genau machen, wie der Ablauf des Programmes ist wird im Folgenden beschrieben.

Alle Programmteile sind selbst Programmiert. Einzig folgende externen Header wurden verwendet: Filestream (fstream.h) für das lesen und schreiben von Textfiles, Input/ Output Stream (iostream.h) für die Ausgabe auf der Konsole, Listen (list.h) um die Ngramme zu speichern und String Stream (sstream.h) für das umwandeln einer Zahl als String zu einem Integer.

Da alles von Grund auf neu programmiert wurde, hat die Erstellung des Programmes viel Zeit in Anspruch genommen. Beim einen oder anderen Programmteil mag nicht die maximale Performance erreicht worden sein. Insbesondere bei der Sortierung und der Suche von Ngrammen gäbe es schnellere Algorythmen, diese jedoch noch zu implementieren hätte den Zeitrahmen dieser Arbeit gesprengt.

Beschreibung des Codes:

-Konstante: n = 3.
Mit dieser Konstante wird die Länge der N-Gramme festgelegt. Die meist verwendete Länge von Ngrammen für die Wortvorhersage ist 3. Das Programm sollte mit jedem beliebigen Wert zwischen 2 und unendlich funktionieren, getestet wurde es aber ausschliesslich für n=3.

-Klasse: Ngram

Besteht aus einem String-Array der Grösse N-1 und einer String-Liste mit Wörtern die an Nter Stelle vorgekommen sind inkl. einer Integer Liste, die angibt wie häufig das entsprechende Wort vorgekommen ist.
Die Idee hinter dieser Gruppierung ist, dass für die Wortvorhersage interessiert, wie häufig auf die N-1 ersten Wörter welches Wort an Nter Stelle vorgekommen ist. D.h. in der finalen Datenbank sollen die aus einem Textkorpus erstellten N-Gramme so geordnet werden, dass für alle N-Gramme bei der die ersten N-1 Wörter gleich sind eine Instanz erstellt wird und das Nte Wort jeweils zur String-Liste dieser Instanz hinzugefügt wird. Für die Wortvorhersage muss dann nur noch die passende Instanz gefunden werden und deren String-Liste konsultiert werden.

Die Klasse Ngram bietet folgende Funktionen:

-void printngram()
Gibt die Werte der Ngram-Instanz in der Konsole aus.

-void sortlasttoken()
Sortiert die String- und Integer- Listen nach ihrer Häufigkeit, d.h. Wörter die häufiger vorkommen sind am Anfang.

Listen von Strings und Ngrammen können mit folgenden Funktionen ausgegeben werden:

-void printstringlist (list<string> stringlist);
Gibt eine String-Liste in der Konsole aus. (z.B. die Wörter die bei bestimmten N-1 vorausgegangene Wörtern an Nter Stelle vorkommen)

-void printngramlist (list<Ngram> ngramlist);
Gibt eine Liste von Ngrammen in der Konsole aus. (z.B. die zur vorhersage verwendete Datenbank)

Für die verschiedenen Programmabläufe werden folgende Funktionen verwendet:

-void writetext(char* path, string text)

Schreibt den Text "text" in ein File am Ort "path".

-list<string> maketokenlist(char* path);
Erstellt aus dem Text am Ort "path" eine Tokenliste und gibt diese zurück. Dieser Tokenizer ist sehr primitiv: die Token werden ausschliesslich durch die Zeichen '.', ' ', '?' und '!' getrennt. Satzende oder zusammenhängende Abkürzungen erkennt der Tokenizer nicht.

-list<Ngram> makengrams(list<string> token);
Erstellt aus einer String-Liste alle möglichen Ngram Instanzen und gibt diese zurück.

-list<Ngram> sortngrams(list<Ngram> ngrams);
Sortiert eine Liste von Ngrammen nach ihren ersten N-1 Token und gibt die sortierte Liste zurück.

-list<Ngram> mergengrams(list<Ngram> ngramsinput);
Fügt in einer sortierten Liste mit Ngrammen, die Ngramme zusammen die die gleichen ersten N-1 Token haben und gibt diese Liste zurück. Funktioniert nur mit sortierten Listen, die genau 1 Element an n-ter Stelle haben.

-void savengrams(list<Ngram> ngrams, char* path)
Speichert die Ngram Liste "ngrams" in einem externen File (Pfad="path") als Textfile. Die ersten N-1 Token jeder Instanz werden mit einem vorhergehenden '!' gekenzeichet. Die Elemente der Liste der Token an n-ter Stelle mit einem '?', die jeweilige Quantität mit einem '.'. Die einzelnen Instanzen werden mit einem ',' getrennt.

-list<Ngram> readdata(char* path)
Liest ein durch 'savengrams' erstelltes Datenfile ein und gibt die Liste der Ngramme zurück.

-float test(list<Ngram> ngrams, list<Ngram> ngramscomp, int nsug)
Testet die Häufigkeit mit der der jeweils letzte Token einer durch einen Text erstellten unsortierten Ngram-Liste ("ngramscomp") durch eine Datenbank von Ngrammen ("ngrams") vorhergesagt werden kann. "nsug" wird hier nicht verwendet.

-float advancedtest(list<Ngram> ngrams, list<Ngram> ngramscomp, int nsug)
Entspricht der Procedure 'test' mit dem Unterschied dass die Anzahl benötigter Tastenanschläge mit und ohne Wortvorhersage berechnet wird. "nsug" ist die Anzahl vorschläge die die dafür berücksichtigt werden (z.B. bei nsug=4 werden stets die 4 wahrscheinlichsten Wörter berücksichtigt).

-list<Ngram> merge (list<Ngram> ngrams, list<Ngram> ngrams2)
Fügt 2 sortierte Ngram-Listen zusammen und gibt die zusammengeführte Liste zurück. Die Listen müssen sortiert sein.

-Ngram search(string searchtoken[n-1], list<Ngram> ngrams, int nsug)
Findet die passende Ngram-Instanz in der Ngram-Liste "ngrams" und gibt die gefundene Instanz zurück. Wird von den Procedures 'test' und 'advancedsearch' verwendet.

-list<string> advancedsearch(string searchtoken[n], list<Ngram> ngrams, int nsug)
Sucht in der Ngram-Instanz die sich mit den searchtoken[0] bis [n-2] Wörtern deckt. Gibt 'nsug' wahrscheinlichsten Wortvorschläge deren Anfang sich mit searchtoken[n-1] deckt zurück.

-Ngram lasttokenmerge (Ngram ngram, Ngram ngram2)
Fügt die Listen für die letzen Token zusammen und gibt die bearbeitete Instanz zurück. Wird von der Procedure 'merge' verwendet.

<u>Ablauf und Funktionsweise der einzelnen Modi:</u>

-Modus: create

Erstellt aus einem Text (Argument 2=Pfad der Datei) eine Ngram-Datenbank und speichert diese in einem Textfile (Argument 3=Pfad der Datei):

```
if (mode=="create"){
token=maketokenlist(textpath);
```

```
ngrams=makengrams(token);
ngrams=sortngrams(ngrams);
ngrams=mergengrams(ngrams);
savengrams(ngrams, savepath);
}
```

-Modus: predict

```
Sagt die wahrscheinlichsten Wörter voraus, die auf den Eingegeben Text(Argument 2)
folgen könnten.
if (mode=="predict"){
ngramsdata = readdata(datapath);
writetext(textpath, text);
token=maketokenlist(textpath);
if (letztes Zeichen == Trennzeichen)
{
kopiere die letzten N-1 Elemente in "stringarray";
result=search(stringarray, ngramsdata, nsug);
printstringlist(result);
}
else if (letztes Zeichen != Trennzeichen)
{
kopiere die letzten N-1 Elemente in "stringarray";
result=advancedsearch(stringarray, ngramsdata, nsug);
printstringlist(result.tokenlast);
}
```

-Modus: test

```
Zählt die Häufigkeit mit der ein Wort im entsprechenden Ngram vorkommt:

if (mode=="test"){
token=maketokenlist(textpath);
ngrams=makengrams(token);
ngramsdata = readdata(datapath);
test (ngramsdata, ngrams, nsug);
}
```

-Modus: advtest

```
Testet die Anzahl eingesparter Tastenanschläge:

if (mode=="advtest"){
token=maketokenlist(textpath);
ngrams=makengrams(token);
ngramsdata = readdata(datapath);
test (ngramsdata, ngrams, nsug);
}
```

-Modus: merge

```
Fügt 2 Datenbanken (Argument 2 und Argument 3) zusammen und speichert diese in
Argument 4:

if (mode=="merge"){
ngramsdata = readdata(datapath);
ngrams = readdata(textpath);
ngramsdata = merge(ngramsdata, ngrams);
savengrams(ngramsdata, savepath);
}
```

AUTHOR

Damian Hiltebrand <damian.h82@gmail.com>