Implementation eines Textgenerators auf Basis eines Markovprozesses

`Autor`: Dennis Lucas Buchholz

1 Einleitung

In dieser Dokumentation wird die Implementation eienr Aufgabe in der Programmiersprache C beschrieben. Ziel dieser Aufgabe ist es, ein Programm zu erstellen, welches eine Datei einliest und ein stochastisches Markov-Model für den Inhalt dieser Datei generiert. Auf Basis dieses stochasitsichen Modells wird anschließend ein neuer Text ausgegeben.

Zur Kompilierung und Testen des Quelltextes wurde der Kompilierer **gcc** in der Version 9.2.0, GNU **make** in der Version 3.82.90 und GNU **ld** (binutils 2.32) aus MinGW verwendet. Für die Erstellung der Dokumentation wurde **texify** von MiKTeX 22.8.28 verwendet.

Hinweis: Für die Erstellung dieses Quelltextes wurde die Bibliothek **utest.h** für Unit-Tests benutzt. Die Erstellung von Unit-Tests stellt keinen Anteil an der Aufgabe dar, jedoch haben die Unit-Tests zur Fehlebehebung und Testung einzelner Programmabschnitte beigetragen. In der Dokumentation wird auf die Unit-Tests nicht weiter eingegangen.

1.1 Verzeichnisstruktur

`/doc`	Alle Dokumentationsdateien
`/include`	Alle Headerdateien (*.h)
`/src`	Alle Quellcodedateien (*.c)
`/test`	Alle Unit-Tests (*.c, *.h)

1.2 Make Befehle

`make`	Alles (Kompiliere Programm und erstelle Dokumentation)
`make shakespeare`	Kompiliere Programm
`make test`	Kompiliere die Unit-Tests und führe sie aus
`make clean`	Alle Objektdateien aufräumen

1.3 Angeforderte Parameter

`-h`	Gebe die Hilfeseite aus
`-f <file>`	Lese die Datei ein
`-s <n>`	Länge des Suchstrings (standardmäßig: 1)
`-l <n>`	Länge der Ausgabe (standardmäßig: Dateigröße in Bytes)

2 Allgemeine Struktur des Programms

2.1 Erkennen der Parameter

Zu Beginn des Programms gilt es zu erkennen, welche Parameter übergeben wurden. Hierfür wird die Funktion `parse_arguments` verwendet, welche mit Hilfe der `getopt` Funktion aus der Bibliothek unistd.h alle Parameter auswertet.

```
1 FILE* file = stdin;
2 char* file_name = malloc(sizeof(char) * 256);
3 file_name[0] = '\0';
4 size_t input_length, output_length;
5
6 parse_arguments(argc, argv, file_name, &input_length, &output_length);
```

An `parse_arguments()` werden die Anzahl der Argumente und ein Array mit den Argumenten übergeben. Des Weiteren werden Zeiger auf den Dateinamen, auf die Suchstringlänge und auf die Ausgabelänge übergeben. Auf den Adressen dieser Zeiger sollen die jeweiligen Werte gespeichert werden.

In der Funktion `parse_arguments()` werden mit Hilfe von `sscanf()` die numerischen Argumente (Suchstringlänge & Ausgabelänge) eingelesen und versucht in einen unsignierten Integer zu konvertieren. Sollte die Konvertierung fehlschlagen, so soll sich das Programm beenden und eine Fehlermeldung ausgeben. Wenn ein Dateiname übergeben wurde soll außerdem überprüft werden, ob sich diese Datei öffnen lässt. Ist dies der Fall, wird der Dateiname an dem Speicherort des Zeigers *file_name* gespeichert. Sollte dies nicht der Fall sein, wird sich das Programm beenden und eine Fehlermeldung ausgeben.

3 Einlesen in einen binären Suchbaum

3.1 Einleitung

Als Datenstruktur für die Speicherung von Suchstrings wurde ein binärer Suchbaum gewählt. Ein Vorteil eines binäreren Suchbaums ist es, dass die Speicherung und Abfrage der Suchstrings immer den gleichen Regeln unterliegt. Es könnte als Beispiel davon ausgegangen werden, dass ein binärer Suchbaum mit dem Wurzelknoten (”Basel”) vorliegt. Ist der in den Funktionsparametern angegebene Suchstring (z.B. ”Aargau”) lexikalisch kleiner als der String des aktuellen Knotens, so wird dieser als linker Unterknoten eingefügt. Ist der angegebene Suchstring (z.B. ”Zürich”) lexikalisch größer als der String des aktuellen Knotens, so wird dieser als rechter Unterknoten eingefügt. Ist der angegebene Suchstring lexikalisch identisch mit dem String des Wurzelknotens, so muss kein neuer Knoten angelegt werden.

3.2 Speichern von Zuständen eines Suchstrings

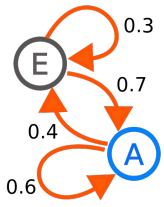
Um einen zufälligen Text zu erzeugen, wird der initiale Suchstring modifiziert. Bei jeder Iteration werden alle Zeichen des aktuellen Suchstrings um ein Zeichen nach links verschoben. Somit fällt das erste Zeichen weg, und an die letzte Stelle des Suchstrings wird das nächste Zeichen in der Datei gesetzt. Wenn als Beispiel der Inhalt einer Datei “Hello World”ist, dann sieht bei einer Suchstringlänge von 5 sieht der Verlauf der Zustände dieses Suchstrings so aus:
‘‘`Hello`’’ →‘‘`ello `’’ →‘‘`llo W`’’ →‘‘`lo Wo`’’ →‘‘`o Wor`’’ →‘‘` Worl`’’ →‘‘`World`’’

Dieses Verfahren wird im Programm nun mit einer Suchstringlänge n mit den Zeichen einer ganzen Datei durchgeführt. Hierzu wird eine Datei in `tree_from_file()` Zeichen für Zeichen ausgelesen und jeder Zustand dieses Suchstrings wird mit Hilfe der Funktion `node_insert()` innerhalb eines neuen Knotens in dem binären Suchbaum gespeichert. Ebenso wird bei jeder Iteration der Buchstabe in dem Knoten gespeichert, welcher auf den Suchstring folgt. Wenn ein Suchstring also mehrfach im Text vorkommt, so wird sich jeder dieser Folgebuchstaben in dem dazugehörigen Knoten gemerkt.

Durch das Speichern der Folgebuchstaben eines Suchstrings kann im weiteren Verlauf mit Hilfe eines Markovprozesses vorausgesagt werden, welcher Zustand des Suchstrings am wahrscheinlichsten folgen könnte.

4 Berechnung des Markovmodells

4.1 Einleitung



(1)

Bei einem Markov-Prozess handelt es sich um einen stochastischen Prozess. Es liegt ein System vor, für das es zu bestimmten Zeitpunkten Zustände gibt und ein Ergebnis, welches immer einen zufällig gewählten Zustand darstellt. Dieser Markov-Prozess enthält Zufallsvariablen, die in Abhängigkeit von bestimmten Annahmen und eindeutigen Wahrscheinlichkeitsregeln von einem Zustand in einen anderen übergehen. Der nächste

Übergang in einen anderen Zustand hängt nur von dem vorherigen Zustand und nicht von der Abfogle der Zustände ab. Diese Annahme erleichtert die Berechnung der bestimmten Wahrscheinlichkeit:

$$\mathrm{P}_{i,j} = P(X_{m+1} = j | X_m = i) \tag{2}$$

4.2 Datenstrukturen für das Markovmodell

Als Datenstruktur für das Markovmodell wird eine Struktur mit der Bezeichnung *MarkovModel* verwendet. Die Struktur enthält einen Zeiger auf die Speicheradresse von dem Suchstring, die Suchstringlänge, die Ausgabelänge und einen Zeiger auf die Speicheradresse des binären Suchbaums mit allen möglichen Zuständen des Suchstrings:

```
typedef struct MarkovModel MarkovModel;
struct MarkovModel {
    char* search_string;
    size_t search_length;
    size_t output_length;
    Tree* tree;
};
```

Die Datenstruktur des binären Suchbaums ist eine Struktur mit der Bezeichnung *Tree*, welche lediglich einen Zeiger auf den Wurzelknoten des Baumes und die Summe aller Knoten beinhaltet. Die Datenstruktur eines einzigen Knotens ist eine Struktur mit der Bezeichnung *Node*, die beinhaltet einen Zeiger auf die Speicheradresse ihres Inhalts, Zeiger auf den linken und rechten Unterknoten, die Länge ihres Inhalts und einen Zeiger auf eine verkettete Liste mit möglichen Folgebuchstaben.

```
typedef struct Node Node;
typedef struct Tree Tree;

struct Node {
    char* content;
    Node* left;
    Node* right;
    size_t length;
    List* list_char;
};

struct Tree {
    Node* root;
    size_t node_len;
};
```

Als Dateistruktur für die Wahrscheinlichkeiten des Markovprozesses wurde eine verkettete Liste verwendet. Für jeden möglichen Folgebuchstaben auf einen Suchstring wird ein Knoten in dieser verketten Liste erstellt. Die Strukturen der verketteten Liste und dessen Knoten sind folgendermaßen deklariert:

```
typedef struct ListNode ListNode;
typedef struct List List;

struct ListNode {
    char character;
    int frequency;
    double probability;
    ListNode* next_element;
};

struct List {
    ListNode* head;
    int sum_of_frequencies;
};
```

In einem Knoten der verketteten Liste wird das Zeichen, die Häufigkeit dieses Zeichens, die Wahrscheinlichkeit dieses Zeichens und ein Zeiger auf den nächsten Knoten erfasst. In der Struktur der Liste wird nur ein Zeiger auf den Kopfknoten und die Summe der Häufigkeiten aller Zeichen erfassen.

Konkret wird die Wahrscheinlichkeit für ein Zeichen folgendermaßen berechnet:

```
void calculate_probabilities(Node* root) {
    ListNode* list_node = root->list_char->head;

    while (list_node != NULL) {
        list_node->probability = (double) list_node->frequency /
                                 root->list_char->sum_of_frequencies;
        list_node = list_node->next_element;
    }

    if(root->left)
        calculate_probabilities(root->left);
    if(root->right)
        calculate_probabilities(root->right);
}
```

Alle Knoten des binären Suchbaums werden im Preorder-Verfahren traversiert, dabei wird für jeden Knoten die verkettete Liste mit den Folgebuchstaben iteriert. Für jeden Knoten in der Liste wird nun die Wahrscheinlichkeit berechnet, indem die Häufigkeit dieses Buchstabens nach diesem Suchstrings durch die Summe der Häufigkeit aller auftretenden Folgebuchstaben dividiert wird.

4.3 Zufällige Auswahl von Knoten mit Wahrscheinlichkeitsverteilung

Auf Basis der Wahrscheinlichkeitswerte jedes Knotens in den verketteten Liste im binären Suchbaum kann nun vorausgesagt werden, welcher Folgebuchstabe am wahrscheinlichsten ist. Diese Zufallsentscheidung ist folgendermaßen implementiert:

```
ListNode* weighted_random(ListNode* head) {
    double s = rand() / (RAND_MAX + 1.0);
    ListNode* list_node = head;
    while (list_node->next_element != NULL
           && ((s -= list_node->probability) >= 0))
        list_node = list_node->next_element;

    return list_node;
}
```

Es wird eine Variable mit einem Zufallswert zwischen 0.0 und 1.0 deklariert, von welcher bei jeder Iteration die Wahrscheinlichkeit eines Knotens der Liste subtrahiert wird. Sobald der Wert dieser Variable kleiner oder gleich 0 beträgt, wird der Knoten ausgewählt, von welchem die Wahrscheinlichkeit zuletzt subtrahiert wurde. Auf diese Weise wurde nun ein Knoten mit einem Folgebuchstaben zufällig, aber auf Basis seiner Wahrscheinlichkeit gewählt.

4.4 Textausgabe auf Basis einer Wahrscheinlichkeitsverteilung

Auf Basis der Wahrscheinlichkeitsverteilung kann nun ein neuer Text erstellt werden. Zunächst wird der initiale Suchstring ausgegeben, welcher sich aus den ersten Buchstaben der Eingabedatei mit der Suchstringlänge n zusammensetzt. Daraufhin wird ein Knoten mit dem Inhalt des Suchstrings in dem binären Suchbaum mit der Funktion `lookup()` gesucht, wird ein solcher Knoten gefunden, so wird ein zufälliger aber wahrscheinlicher Buchstabe mit Hilfe von `weighted_random()` gewählt und asugegeben.

Ebenso werden alle Zeichen des Suchstrings um eine Position nach links verschoben und der gewählte Buchstabe wird an den Suchstring angehangen. Dieser Prozess wird wiederholt, sobald genau so viele Zeichen ausgegeben wurden, wie mit der Ausgabelänge m angegeben. Konkret wurde dies folgendermaßen implementiert:

```
char next_letter(MarkovModel *state) {
    Node* n = lookup(state->tree->root, state->search_string,
                     state->search_length);
    if(!n) {
        model_destroy(state);
        fprintf(stderr,
                "Error: Unable to make prediction for search string '%s'",
                state->search_string);
        exit(EXIT_FAILURE);
    }
    return weighted_random(n->list_char->head)->character;
}
void generate_text(MarkovModel *model) {
    char last_letter;
    srand((unsigned int) time(NULL));

    printf("<%s>", model->search_string);

    for (int i = 0; i < (int) model->output_length; i++) {
        last_letter = next_letter(model);
        printf("%c", last_letter);
        shift_string(model->search_string, (int) model->search_length);
        model->search_string[model->search_length - 1] = last_letter;
    }
}
```

Wenn die Beispieldatei “7zara10.txt” aus der Aufgabe eingelesen wird, kann die zufällige Textausgabe des Programms demonstriert werden:

```
$ ./shakespeare -s 19 -l 600 -f 7zara10.txt
<Friedrich Nietzsche>

Also sprach Zarathustra. Der Heilige antwortete: Ich mache Lieder und singe sie,
 und wenn ich Lieder mache, lache, weine und brumme ich: also lobe ich Gott.
Mit Singen, Weinen, Lachen und Brummen lobe ich den Gott, der mein Gott ist.
Doch was bringst du uns zum Geschenke?
Als Zarathustra diese Reden sprach, stand er nahe dem Eingange seiner Hoehle lag.
Aber, indem er mit den Haenden um sich und ueber sich und unter sich griff, und
den zaertlichen Voegeln wehrte,
siehe, da geschah ihm etwas noch Seltsameres: er griff naemlich dabei unvermerkt
 in ein dichtes warmes Haar-Gezottel hinein;
```

Bei erneutem Einlesen ist eine andere Ausgabe ersichtlich:

```
$ ./shakespeare -s 19 -l 600 -f 7zara10.txt
<Friedrich Nietzsche>

Also sprach Zarathustra. Zwar ihr sagt: "die Lust an kleinen Bosheiten erspart
 uns manche grosse boese That." Aber hier sollte man nicht sparen wollen.
 Wie ein Geschwuer ist die boese That: sie juckt und kratzt und bricht heraus,
 - sie redet ehrlich. "Siehe, ich bin Krankheit" - so redet die boese That;
 das ist ihre Ehrlichkeit.
Aber dem Pilze gleich ist der kleine Gedanke: er kriecht und duckt
```