Veranstaltung von: Prof. Martin Hering-Bertram / Dr. Lars Prädel
Hochschule Bremen

Projekt 1: Datenkompression

Medieformate und Algorithmen

Stefan Bieliauskas & Jennifer Theloy
11.11.2014

Inhalt

Übungsblatt 1

Allgemeine Änderungen an der Klasse

Zuerst haben wir die Datentypen von *int* auf *char* geändert. Der Vergleich funktionierte weiter, da die *chars* nach ihrem ASCII Wert sortiert werden und die ASCII-Zeichen bereits lexikalisch sortiert sind.

Danach wurde die **TreeNode::Print** Methode angepasst, damit dieser immer die übergeordnete Ebene übergeben wird. Für die Inverse Darstellung des Baumes musste die Zeile 109 angepasst werden sodass dort anstatt ein < ein > in der IF-Anweisung steht.

Übungsblatt 2

Transform

Private:

Double tx // Die X-Koordinate des zu transformierendes Punkts
Double ty // Die Y-Koordinate des zu transformierendes Punkts
Long double alpha // Der Winkel zur Rotation

Public:

Transform(long double alpha, double tx, double ty) // Konstruktor
std::array< double, 2 > & Apply(double *x, double *y) // rotiert und translatiert den Punkt (x,y)
std::array< double, 2 > & ApplyInverse(double *x, double *y) // // wendet die inverse Transformation an

Main

In der Main befindet sich das angeforderte Testprogramm. Anhand von vier Beispielen wird vergleichen, ob die berechneten Ergebnisse der Transform-Klasse mit den realen Ergebnissen übereinstimmen.

Übungsblatt 3 & 4

Entropie

Konstanten:

int LAST_ASCI_CODE = 255 // Stellt den Indes des letzten ASCII Zeichens dar.

Private:

vector<unsigned char>* vec // Vektor, der die Chars des Textes enthält, von dem die Entropie berechnet werden soll.

Public:

Entropie(vector<unsigned char> *vec) // Konstruktor
double getEntropie() // Berechnet die Entropie

double getBestFileSize() // berechnet die optimale Größe des Files

Double getBestFileSize()

Berechnet die optimale Länge eines Textes indem die Länge des ursprünglichen Textes mal die Entropie genommen wird.

Double getEntropie()

Zuerst wird in einer *For*-Schleife der *Char* Vector durchlaufen und die Häufigkeit jedes einzelnen Zeichens ermittelt. Diese Häufigkeiten werden in einen leeren *Vector* geschrieben an die Stellen entsprechend der *ASCII* Zahl des jeweiligen Zeichens.

Dieser *Vector* wird nun durchgegangen und von jedem Zeichen mit einer angegeben Häufigkeit wird die Wahrscheinlichkeit und der Entscheidungsgehalt berechnet, welche mit einander multipliziert werden. Das Ergebnis wird dann jeweils immer auf ein *Double* addiert, sodass man am Ende der *For*-Schleife die Entropie erhält, welche zurückgegeben wird.

LZW-Codierung & -Decodierung

Konstanten:

int LAST_ASCI_CODE = 255 // Stellt den Indes des letzten ASCII Zeichens dar.
int START_INDEX_OF_MUSTER = LAST_ASCI_CODE +1 // Der Index, ab welchem die Muster eingetragen werden können.
int MAX_INDEX_OF_MUSTER = 1023 // Der Index, bis zu dem maximal Muster eingetragen werden können.

Private:

map<int,vector<unsigned int>> muster // Eine Map in der die Muster zugehörig zu ihrem Index abgespeichtert werden.
int indexOfNewMuster // Der Index des Muster, das als nächstes eingetragen wird.
bool encodeMuster(vector<unsigned int> searchString, int* outputMusterId) // Codierung
void insertMuster(vector<unsigned int> newMusterString) // Fügt ein Muster in die Map .
bool decodeMuster(int searchID, vector<unsigned int> *outputVector) //Decodierung.
void setDefaults() // Setzt die Standards.

Public:

LZWClass () // Konstruktor
vector<char> decode(vector<unsigned int> *vec) // Codierung
vector<unsigned int> encode(vector<unsigned char> *vec) // Decodierung

Bool decodeMuster()

In dieser Methode soll ermittelt werden, ob einer bestimmten ID (*searchID*) bereits ein Muster zugewiesen ist. Dabei wird die *Map* mit den Mustern durchgegangen und ein Wahrheitswert zurückgegeben. Wenn eine ID gefunden wurde wird an den *outputString* das nächste *Char* (entsprechend des *unsigned Int* in der Map) angefügt, welches das erste des gefundenen Musters darstellt.

Bool encodeMuster()

In der Funktion wird ermittelt, ob der übergebene *String* bereits als ein Muster abgespeichert ist. Sollte ein Muster gefunden worden sein wird die *outputMusterId* neubestimmt als die ID

des gefundenen Musters. Auch hier wird ein Wahrheitswert zurückgegeben der variiert je nachdem, ob die Suche erfolgreich war oder nicht.

Void insertMuster()

Die Methode dient dazu ein Muster zu der *Map* hinzuzufügen. Sollte der Index des neuen Musters den Maximalen Index überschreiten wird damit begonnen Muster zu überschreiben indem *indexOfNewMuster* auf den Startindex zurückgesetzt wird (*START_INDEX_OF_MUSTER*).

vector<char> decode()

In dieser Methode wird ein codierter Text wieder decodiert. Dafür wird ein übergebener *Int Vector* mit dem codierten Text übergeben. Es wird jedes einzelne *Int* durchgegangen und wenn dieses unter 256 ist wird es sofort in ein *Char* umgewandelt und an den *Ouput* angefügt. Sollte die Zahl über 255 sein wird zuerst mit Hilfe von *decodeMuster* das Muster ermittelt und dann an den Output angefügt.

Danach wird das neue Muster in die Muster *Map* eingefügt, indem sich das nächste *Char* über den nächsten Index angeschaut wird. Sollte es ein ASCII Zeichen sein wird dieses einfach in ein *unsigned Int* entsprechend des Index des ASCII Zeichens umgewandelt und agefügt, sollte es sich jedoch um ein Muster handeln wird dieses zunächst einmal ermittelt. Dann wird das erste *unsigned Int* Index des Musters entnommen und an *newMuster* angefügt.

Zum Schluss wird der Output zurückgegeben.

vector<int> encode()

In dieser Methode soll ein Text codiert werden. Es wird ein dabei in einer *For*-Schleife über den übergebenen *unsigned Int Vector* gegangen. Dabei wird solange das nächste *unsigned Int* (also der entsprechende Index des Zeichens) angeschaut, bis kein Muster mehr in der *Map* gefunden wurde. Dann wird das nicht gefundene Muster in die *Map* hinzugefügt und der Index des zuletzt gefundenen Musters (Beziehungsweise ASCII-Zeichens) dem Output hinzugefügt.

Zum Schluss wird der Output zurückgegeben.

BitStream

Public:

BitStream () // Konstruktor
void saveBinaryFile(vector<unsigned int>* vec,string fileName) // Speichert ein Binäres File ab.
vector<unsigned int> readBinaryFile(string fileName) // Liest das Binäre File wieder ein.
vector<bool> getBinary(vector<unsigned int>* vec) // Wandelt einen vector<unsigned int> in einen boolischen *Vector* um.
vector<unsigned int> getVector(string* bitset) // Wandelt einen String mit Nullen und Einsen (Bitfolge) in einen vector<unsigned int> um.

Void saveBinaryFile()

Um ein Binary File abzuspeichern muss zunächst ein *Vector* mit *Booleans* angelegt werden, indem die die Sequenzfolge der Bits des Textes abgespeichert werden. Danach wird das File

geöffnet, in welches gespeichert werden soll. Danach muss die Anzahl der benötigten Bytes berechnet werden, um dann ein *Char Array* zu erzeugen, welches so viele Stellen hat, wie Bytes benötigt werden.

Als nächstes werden mit Hilfe der Hilfsklasse *Struct_bit* die *Chars* in dem Array erzeugt entsprechend der Bitfolge des *boolean Vectors*. Dieses Array wird in dann in die Datei abgespeichert.

Vector<unsigned int> readBinaryFile()

Die Funktion lädt das entsprechende File und wandelt die *Chars* zurück in eine entsprechende Bitfolge.

Main

Public:

void executeFile(string fileName, bool debug) // Dateien werden geladen und jeweilige Funktionen werden aufgerufen.
int main(int argc, char *argv[]) // Main

void executeFile()

Die Funktion öffnet zuerst die einzulesende Datei und unterscheidet dabei zwischen verschiedenen Dateiformaten. Anschließend werden die jeweils für die Entropie, die LZW-Codierung und –Dekodierung und für die Bitstream Speicherung ausgeführt und jeweils mit erklärenden Konsolenausgaben Vervollständigt um die Ausgaben übersichtlicher zu gestalten.

Testläufe

Testlauf 1

Im folgenden Testlauf wurde das Beispiel aus der Vorlesung verwendet (MedForm1 14. 10.'14) und in diesem Fall wurden die einzelnen Schritte für die Entropie, die LZW-Codierung und –Decodierung und die Bitstream Speicherung ausführlich angezeigt.

Hierbei ist zu beachten, dass es bei der Binären-Speicherung des LZW-Textes in 10 Bit-Schritten zu einem Bit als Überhang kommen kann, da c++ nur 8 Bit am Stück schreiben kann. Dieser fall tritt auf, wenn die Anzahl der LZW-Codierten Zeichen nicht glatt durch 8 Teilbar ist.

========Datei: Text_Short_LZWLW78.txt=======
Entropie für die Datei: 2.8812 Bits / Zeichen
Optimale Länge der Datei : 8.28344 Bytes

====LZW===
Original-Länge: 23 Zeichen (23 Bytes)
0. Original: L(76)
1. Original: Z(90)
2. Original: W(87)
3. Original: L(76)
4. Original: Z(90)
5. Original: 7(55)
6. Original: 8(56)
7. Original: L(76)
8. Original: Z(90)
9. Original: 7(55)
10. Original: 7(55)
11. Original: L(76)
12. Original: Z(90)
13. Original: C(67)
14. Original: L(76)
15. Original: Z(90)
16. Original: M(77)
17. Original: W(87)
18. Original: L(76)
19. Original: Z(90)
20. Original: A(65)
21. Original: P(80)
22. Original: LF(10)

Encodieren-Länge: 17 Zeichen (21.25 Bytes)
0. Encode: L(76)
1. Encode: Z(90)
2. Encode: W(87)
3. Encode:
4. Encode: 7(55)
5. Encode: 8(56)
6. Encode: _(259)
7. Encode: 7(55)
8. Encode:

9. Encode: C(67)
10. Encode:
11. Encode: M(77)
12. Encode: _(258)
13. Encode: Z(90)
14. Encode: A(65)
15. Encode: P(80)
16. Encode: LF(10)

Decodeieren-Länge: 23 Zeichen
0. Decode:L(76)=Original:L
1. Decode:Z(90)=Original:Z
2. Decode:W(87)=Original:W
3. Decode:L(76)=Original:L
4. Decode:Z(90)=Original:Z
5. Decode:7(55)=Original:7
6. Decode:8(56)=Original:8
7. Decode:L(76)=Original:L
8. Decode:Z(90)=Original:Z
9. Decode:7(55)=Original:7
10. Decode:7(55)=Original:7
11. Decode:L(76)=Original:L
12. Decode:Z(90)=Original:Z
13. Decode:C(67)=Original:C
14. Decode:L(76)=Original:L
15. Decode:Z(90)=Original:Z
16. Decode:M(77)=Original:M
17. Decode:W(87)=Original:W
18. Decode:L(76)=Original:L
19. Decode:Z(90)=Original:Z
20. Decode:A(65)=Original:A
21. Decode:P(80)=Original:P
22. Decode:LF(10)=Original:

Compression: 26.087%

====Bitstream-Speicherung====

Binärer String:
00010011000001011010000101011101000000000000110111000011100001000000110000110111010000000000010000110100000000000100110101010000001000010110101000010000010001010000000000001010
Binäre save: 17 Zeichen
Binäre Datei save: Text_Short_LZWLW78.txt.bin
Binäre Datei load: Text_Short_LZWLW78.txt.bin
Binäre load: 18 Zeichen
0. BinarDecode: 76 Original: 76
1. BinarDecode: 90 Original: 90
2. BinarDecode: 87 Original: 87
3. BinarDecode: 256 Original: 256
4. BinarDecode: 55 Original: 55
5. BinarDecode: 56 Original: 56

6. BinarDecode: 259 Original: 259
7. BinarDecode: 55 Original: 55
8. BinarDecode: 256 Original: 256
9. BinarDecode: 67 Original: 67
10. BinarDecode: 256 Original: 256
11. BinarDecode: 77 Original: 77
12. BinarDecode: 258 Original: 258
13. BinarDecode: 90 Original: 90
14. BinarDecode: 65 Original: 65
15. BinarDecode: 80 Original: 80
16. BinarDecode: 10 Original: 10

Testlauf 2

Im zweiten Testlauf wurde ein längerer Beispieltext verwendet um eventuell auftretenden Fehler zu erkennen.

========Datei: Test_Long_Lorem.txt=======
Entropie für die Datei: 4.16293 Bits / Zeichen
Otimale länge der Datei : 156546 Bytes

====LZW===
Original-Länge: 300838 Zeichen (300838 Bytes)
Encodieren-Länge: 118398 Zeichen (147998 Bytes)
Decodeieren-Länge: 300838 Zeichen
Compression: 60.6439%

====Bitstream-Speicherung====
Binäre save: 118398 Zeichen
Binäre Datei save: Test_Long_Lorem.txt.bin
Binäre Datei load: Test_Long_Lorem.txt.bin
Binäre load: 118399 Zeichen

Testlauf 3

Im dritten Testlauf wurde die Bilddatei Lena.ppm verwendet. Bei der Codierung oder Decodierung treten bei den letzten Beiden Zeichen Fehler auf. Es kann jedoch nicht an der Länge der Datei liegen, da im zweiten Testlauf eine wesentlich längere Datei verwendet wurde und der Fehler nicht aufgetreten ist (Fehlerrate bei 1,017174999872853e-5%).

========Datei: Test_lena.ppm=======
====LZW===
Original-Länge: 196623 Zeichen (196623 Bytes)
Encodieren-Länge: 92830 Zeichen (116038 Bytes)
Decodeieren-Länge: 196621 Zeichen
196621. Fehler:An dem Char mit dem Index: 196621 wurde kein Ergebnis der LZW-Codierung gefunden. Es fehlte das Zeichen: 98(b)
196622. Fehler:An dem Char mit dem Index: 196622 wurde kein Ergebnis der LZW-Codierung gefunden. Es fehlte das Zeichen: 98(b)

Compression: 52.7878%

====Bitstream-Speicherung====
Binäre save: 92830 Zeichen
Binäre Datei save: Test_lena.ppm.bin
Binäre Datei load: Test_lena.ppm.bin
Binäre load: 92831 Zeichen

Testlauf 4

In diesem Testlauf wurde die Translation und Rotation getestet. Wie sich zeigt, wurde durch den Cast von *Double* nach *Float* ein kleiner Rundungsfehler eingebaut, sodass die Werte nicht als komplett identisch angesehen werden, obwohl sie in der Theorie korrekt berechnet werden.

3. Fehler Apply: expeceted: (2.666667461395263671875,4.833333492279052734375)

actual: (2.666667461395263671875,4.833333492279052734375)

3. Fehler Apply-Inverse: expeceted: (0.833333313465118,0.333333343267441)

actual: (0,0)

2. Apply OK

2. Fehler Apply-Inverse: expeceted: (0.5,1.5) actual:(-6.12323426292584e-17,1)

1. Apply OK

1. Fehler Apply-Inverse: expeceted: (3,4)

actual:(3,4) 0. Apply OK

0. Fehler Apply-Inverse: expeceted: (4,7)

actual:(4,7)