Information Retrieval 2
Zusammenfassung

Benedikt Lüken-Winkels

9. Juli 2018

Inhaltsverzeichnis

1 Schriftsysteme, Zeichen und Alphabete

1.1 Unicode

Ziele des Unicodedesigns

- Universell einsetzbar: alle Zeichen können verwendet und dargestellt werden
- Effizient: Einfacher Text ist leicht zu parsen, während komplizierte Zeichen eindeutig und schnell decodiert werden
- Eindeutig: Jedes Zeichen hat genau einen Unicode code point.

Designprinzipien von Unicode

- Universality
- Efficiency
- Characters, not glyphs: Zeichen werden auf Glyphen gemappt
- Semantics
- Plain text
- Logical order
- Unification: verschiedene Zeichen mit der gleichen Form, Nutzung und Eigenschaften aus verschiedenen Sprachen erhalten den gleichen code point
- Dynamic composition: Zeichen, wie Ü werden aus dem code point für U und dem non-spacing mark 'Anführungszeichen'
- Equivalent sequences
- Convertibility

1.1.1 UTF: Unicode Tansformation Formats

Mit Unicode 2.0 und UTF-16 werden alle Zeichen außerhalb des Basic mulitlingual plane durch high und low surrogate abgebildet. Auch leading und trailing surrogates.
high surrogate: 1024 code points
low surrogate: 1024 code points

UTF-16 Kodierung eines Wortes außerhalb der BML in zwei Teilen:

1. Abziehen von 65536 (10000_{hex}) $\Rightarrow$ 20Bit Zahl im Bereich 00000_{hex} - $FFFFF_{hex}$ beleibt übrig
2. Aufteilen in 2 10 Bit Blöcke
3. Dem ersten Block wird 110110 vorne angestellt, dem zweiten 110111: Erster Block = High surrogate, Zweiter Block = Low surrogate

Java und Unicode Zwei Identifier sind gleich, sobald sie den selben Unicode-Charakter für jedes Zeichen haben. Sie können also gleich aussehen, aber trotzdem verschieden sein. Zum Beispiel a aus verschiedenen Schriftsystemem (griechisch und lateinisch)

Combining Class Diakritische Zeichen können in fast beliebiger Reihenfolge codiert werden $\Rightarrow$ Jedes Zeichen erhält einen Wert zwischen 0 und 255. Anhand dieser Zahl wird die Reihenfolge der code points bestimmt, sodass das Zeichen eindeutig kodiert wird.

2 Textdokumente

Postscript

- Turingvollständige Programmiersprache
- Stackorientiert: Postfix Operatoren / reverse polish notation
- Teilweise Verwendung als Zwischenschritt bei Druckern etc
- Problematisch bei maschineller Texterkennung, da fließender Übergang zwischen graphischen Elementen und Textzeichen

⇒ Überschreiben von Operatoren, wie 'show', sodass die Elemente in einer eigenen Datenstruktur gespeichert werden.

PDF

- 7Bit ASCII Datei, **keine** Programmiersprache, also keine Schleifen oder If-statemets usw.
- Ähnlichkeiten zu Postscript: graphische Primitive
- Aufbau in Hierarchien in der Datenstruktur

⇒ Logische Struktur des Dokuments darstellbar

- Elemente:
  - Dictionaries: Name-Wert-Paare
  - Arrays: lineare Liste ohne Typen
  - Booleans, Numbers, Strings, Names
  - Streams: Anfang, Ende und Länge sind Tags im Dokument
- Aufbau:
  - Header: Codierung und Dekodierverfahren
  - Body
  - Cross-Reference Table: Enthält Zeiger für alle indirekten Objekte
  - Trailer
- **OCR** Optical Character Recognition extrahiert Text aus Bildern und erzeugt positionierten Text, der das Bild ersetzt.

3 Exakte Suche in Strings

3.1 Knuth-Morris-Pratt-Algorithmus

Vorverarbeitung des Suchmusters:

$$\begin{aligned} b &\in \{0...k-1\} \\ Suchmuster\ x &= x_0...x_{k-1} \\ echtes\ Suffix\ u &= u_{k-b}...u_{k-1} \\ echtes\ Praefix\ u &= u_0...u_{b-1},\ also\ x \neq u \\ Rand\ r &= x_0...x_{b-1} \wedge r = x_{b-k}...x_{k-1} \end{aligned}$$

Fortsetzung des Randes r durch a, falls ra Rand von xa

- Erstellung eines Arrays d mit Länge(d) = k+1, $d[i]$ ist die Länge des breitesten Randes für jedes Präfix der Länge i. Ist die Länge des Präfixes 0, so wird im Array der Wert -1 eingetragen.
- Prüfe, ob sich ein Rand von $b[m-1]$ durch α fortsetzen lässt. Wenn ja, $b[m] = b[m-1] + length(\alpha)$

Suche:

- Fange an, Zeichen für Zeichen zu vergleichen. Bei Mismatch, verschiebe um $|b[i] - i|$, sodass Suffix und Präfix überlappen. Dieser Teil muss nicht mehr überprüft werden.

Laufzeit: O(n) + O(m)

- n Länge des Dokumentes
- m Länge des zu durchlauf Suchstrings

3.2 Boyer-Moore-Algorithmus

Vorverarbeitung des Suchmusters
Sprungtabelle

- Bad-Character: Abstand zwischen dem letzten Vorkommen jedes Zeichens im Suchmuster zum Ende des Musters
- Good-Suffix: Abstand zwischen jedem Teilmuster

Suche:

- Lege das Muster linksbündig an den Suchstring an und vergleiche die Zeichen von rechts nach links. Bei Mismatch, die maximale Verschiebung von:

- **Bad-Character-Heuristik:** Verschiebung des Musters, bis Mismatchzeichen im Suchstring und letztes Vorkommen des Zeichens im Muster übereinanderliegen. Bei Verschiebung nach links: ein Feld nach rechts. Zeichen $\notin$ Muster: Komplett verschieben.
- **Good-Suffix-Heuristik:** Stimmt ein Suffix des Musters überein, wird das Muster bis zum nächsten Vorkommen des Suffixes im Muster verschoben oder komlett.

Laufzeit:

- Günstigster Fall: Beim ersten Vergleich wird ein Zeichen gefunden, das nicht Muster vorkommt. O($\frac{n}{m}$)

4 Vorverarbeitung von Dokumenten

- Suchmaschinen gehen von statischen Dokumenten aus

$\Rightarrow$ ermöglicht Indexierung

Indexe

- Datenbank besteht aus Sammlung von Dokumenten. $\rightarrow$ Definition eines Dokuments wichtig (Seite, Text, Wort, ...)
- Vorverarbeitung von Termen:
  - Stemming
  - Case-folding (zB Kleinschreibung)
  - Auslassen von Stoppworten

4.1 Invertierte Dateien

- Einträge der Form (Term, Zeigerliste auf alle Vorkommen)
- Indexkompression nötig, da zu viel Speicher benötigt

4.1.1 Differenzdarstellung

$$< t, f_t, [d_1, d_2, ..., d_{ft}] > t = term, f_t = number(docs)$$
$$< t, f_t, [d_1, d_2 - d_1, ..., d_{ft} - d_{ft-1}] >$$

$\Rightarrow$ zu wenig Kompression

4.1.2 γ-Kode

Anzahl der folgenden übrigen Bits als Präfix im Unärcode $\Rightarrow$ Vorderste 1 wird abgeschnitten

γ-Kode	Binär
10 0	10
10 1	11
110 00	100
...	...
1110 000	1000

4.1.3 δ-Kode

Anzahl der folgenden Bits im Präfix im γ-Kode

δ-Kode	Binär
100 0	10
100 1	11
101 00	100
...	...
1101 000	1000

4.1.4 Globales Bernoulli-Modell/Golomb Code

Darstellung der Zahl n durch Quotienten q und Rest r, der bei der Division durch Parameter b übrig bleibt. Mit dem Steuerungsparameter c wird die Ausgabe codiert. c ergibt sich aus dem aufgerundeten $\log_2 b$.
Zunächst werden $q + 1$ 1en gefolgt von einer 0, dann wird abhängig von c entweder:

- r als Binärcode mit $|[r]_2| = c - 1$, falls $r < 2^c - b$ oder
- $r + 2^c - b$ als Binärcode mit $|[r + 2^c - b]_2| = c$ geschrieben.

Vorteile Je größer der Parameter, desto langsamer wächst die Anzahl der zur Darstellung benötigten Bits, aber desto größer ist die Anzahl der minimal benötigten Bits für die kleinen Zahlen.

Voraussetzung Geometrische Verteilung der zu kodierenden Daten nötig

4.2 Invertierte Dateien für das Vektorraummodell

Idee Zusätzliche Speicherung der Vorkommen eines Terms innerhalb eines Dokuments.

Aufbauschritte

1. Aufteilen des Texts
2. Erstellen einer Häufigkeitsmatrix
3. Übertragen in Tupel (Transponieren der Matrix)

4.2.1 Memory-based inversion

Idee Erstellen einer Hash-Tabelle

4.2.2 Sort-based inversion

Idee Indexierung bei großen Dokumenten nur effizient, wenn sequenziell gearbeitet wird.

1. Erstellung eines Tupels für jeden Term in den Dokumenten ¡Term(-nummer), Dokument, Vorkommen von Term in Dokument¿
2. Sortierung der Tupel nach Dokument und Aufteilung in Blöcke
3. Mergesort nach Termnummer

Sortierverfahren

- Quicksort: am schnellsten, aber genügend Hauptspeicher notwendig
- Mergesort schnelles und stabiles Sortierverfahren, wenig Hauptspeicher nötig

$\Rightarrow$ Quicksort, soglange genug Hauptspeicher, dann Mergesort

4.2.3 Kanonischer Huffman code

Idee Präfixfreie Kodierung der Terme $\rightarrow$ je seltener das Wort, desto Länger die Kodierung. Worte gleicher Länge sind alphabetisch sortiert

Besonderheiten des kanonischen Huffman codes Nach der Bestimmung der Codewortlänge werden die Terme innerhalb der Längenklassen sortiert und erhalten dann aufsteigende (Binär-)Codierungen.

Huffman-Baum zur Bestimmung der Codelänge Erstellung eines Huffman-Baums für Wort der Länge n

1. Erstellen eines Arrays mit 2n Einträgen (0 bis n-1: Zeiger auf Heap; n bis 2n-1: Häufigkeiten)
2. Bauen eines Min-Heaps (Kindknoten sind größer)
3. Iterieren
   (a) Entfernen der Wurzel r_1, Indizes der Pointer rücken nach links
   (b) Entfernen der Wurzel r_2, erstes Feld bleibt frei
   (c) Schreibe die Summe der Häufigkeiten an die hinterste Stelle der Pointer. Die Stellen der Häufigkeiten werden durch Pointer auf die Summe ersetzt. Schreibe einen Pointer auf die Summe an die Stelle des Pointers von r_2
   (d) Stelle Min-Heap wiederher
4. Tiefe des Blattes im neuen Heap ist die Länge des Codewortes

Problem Degenerierter Baum bei ungerader Anzahl an Blättern/Knoten $\Rightarrow$ Einführen von Dummyknoten nötig

4.3 Signatur-Dateien

Idee Terme werden gehasht und dann durch Superimposing (Überlagerung) aneinandergehangen.

- Test ob alle Einsen der Anfragesignatur gesetzt sind
- Zu viel Überlagerung sorgt für Ungenauigkeit und vielen false Drops
- Bei **vertikaler Partitionierung** werden nur die 'Signaturscheiben' genommen, wo die Anfragebits 1 sind (?)

5 Edit-Distanz, Approximative Suche

5.1 Phonetische Verfahren

Soundex Ähnlichkeit durch Normierung bestimmen (Löschen der Vokale, Mehrere Zeichen haben den gleichen Code)

Phonix Soundex Variante, bei der Buchstaben transformiert werden

5.2 String-Ähnlichkeit

5.2.1 Levenshtein-Metrik/Edit-Distanz

Idee Abstand d zwischen zwei Worten, falls d Operation nötig sind, um das eine in das andere Wort zu überführen.

Substring-Suche Setze obere Zeile auf 0: Kommt unten eine 0 an, Substring gefunden

Ukkonen Cut-off Abbrechen der Berechnung, sobald die Fehlertoleranz erreicht ist. Dafür: Initialisierung der ersten Zeile mit 0

5.2.2 Landau-Vishkin (brute-force

Idee Finden aller Vorkommen des Suchpatterns mit dem Editabstand $\leq$ k mit Hilfe von Suffix-Bäumen

5.2.3 Shift-And-Algorithmus

Idee Suchanfragen sind nicht länger als ein Maschinenwort (64 Bit). Dadurch Parallelisierung der Bit-Operationen ($\Rightarrow$ Beschleunigung)

Verfahren

1. Erstelle Bit-Vektor mit Position des Zeichens für jeden Buchstaben im Suchmuster. (Einige können zusammengefasst werden)
2. Vergleich mit Text:
   (a) SHIFT rechts
   (b) OR mit 100...00
   (c) AND mit Bit-Vektor des Zeichens
3. Kommt eine 1 durch, bzw 'unten an' $\Rightarrow$ Match

Erweiterung Weitere Arrays R_j^d kodieren Arrays mit d Fehlern. Verknüpfung bedeutet linearen Mehraufwand $\Rightarrow$ Levenshtein auf Bit-Ebene runterbrechen und Bit-parallel zu machen

Verwendung Wird bei **agrep** verwendet. Platz O(m), Zeit O(n)

5.2.4 Längste gemeinsame Teilsequenz LCS

Idee Wenige Operationen um von String1 zu String2 zu kommen.

Verfahren

1. Baue Matrix auf: 1. Spalte und 1. Zeile := 0
   - bei gleichen Zeichen ++
   - sonst Maximum von oben und links
2. Backtracking der Lösung

$\Rightarrow$ Diagonale bedeutet Teilsequenz

Verbesserung Naiv sehr hoher Platzbedarf. Durch Monotonie der Matrix in Spalte, Zeile und Diagonale müssen nur Eckpunkte kodiert werden.

6 Suffix-Bäume

Idee Wird derselbe Text immer wieder durchsucht, ist es sinnvoll eine Hilfsdatenstruktur aufzubauen.

Trie Enthält alle möglichen Wege die Worte zu erstellen: Ein Knoten für jeden Buchstaben des Alphabets

Suffix-Trie Enthält alle möglichen Suffixe des Dokuments: Ende eines Suffixes wird durch $\$_1 ... \$_n$ für n Dokumente beschrieben.
Hoher Platzaufwand $\Rightarrow$ kompakter **Patricia Trie:** linearer Platzaufwand, da Daten komprimiert dargestellt werden und nicht jedes Zeichen einen Knoten bekommt. Zudem kann die Kantenbeschriftung durch Position und Länge des Teilstrings erfolgen
Realisierung:

- falls $|\Sigma|$ groß: Hashtabelle für den Baum
- falls $|\Sigma|$ klein: Array
- balancierte Bäume

6.1 Approximate String Matching

Idee Baumverzweigung für jede Spalte der Edit-Distanz. Tiefe wird durch Fehlertoleranz beschränkt

Impliziter Suffix-Baum Streichen der Endzeichen aus einem Suffix-Baum, Zusammenfassen aller Knoten mit Ausgangsgrad kleiner 2

Suffix-Erweiterung Erweiterung eines vorhandenen Suffixes β um ein neues Zeichen S(i+1). 3 Fälle sind zu unterscheiden:

1. β endet in einem Blatt: Erweiterung um ein Zeichen (neues Blatt) βS(i+1)
2. β endet nicht in einem Blatt und es gibt eine weiterführende Kante, die nicht S(i+1) ist
   (a) Neues Blatt: Kante wird mit βS(i+1) beschriftet
   (b) β endet in mitten eine Kante: Kante wird gesplittet und ein neues Blatt S(i+1) eingefügt
3. βS(i+1) ist bereits im Baum

Suffix-Links Verlinkung innerhalb des Baums. Sobald es einen Teilstring gibt, der bereits im Baum vorgekommen ist, wird dieser dem Baum angehangen und eine Verlinkung in Form eines Eintrags in eine lineare Liste zum vorherigen Vorkommen eingetragen.