Inhaltsverzeichnis

1 Allgemeines

1.1 Klausurthemen

- Suffix-Bäume
- Aho-Corasick, Hirschberg
- Turingmaschinen, Automaten
- Dynamische Programmierung
- nicht: NP-Problem, klassiche PM-Algorithmen (Boyer-Moore, etc.)

1.2 Bemerkung

Das hier sollte ursprünglich eine Gedächtnisstütze für mich werden, natürlich ohne Anspruch auf Vollständigkeit. Habe mir die aus meiner Sicht wichtigsten Sachen aus dem Gusfield, meiner Mitschrift, Adrians Tutorium und mit eigenen Worten zusammengefasst. Das Ganze ist nachher deutlich länger geworden als geplant. Fehler bitte an trieglaf@inf.fu-berlin.de

2 Pattern-Matching

- Σ endl. Alphabet, P, T aus Σ, $|P|$ = n und $|T|$ = m
- naiver Ansatz: $O(nm)$ (alles durchtesten...)
- Untere Schranke: $\Omega(n+m)$ (Text und Pattern lesen)

2.1 Aho-Corasick (Gusfield S. 52)

Wichtiges:

- Laufzeit $O(n+m+k)$ (k = Anzahl der Matches, Laufzeit ist outputsensitiv)
- Datenstruktur ist ein Keywordbaum, d.h. ein gerichteter Baum κ mit Wurzel r. Jede Kante ist mit genau einem Buchstaben (Label) beschriftet. Kanten, die von einem gemeinsamen Knoten ausgehen, sind mit unterschiedlichen Label beschriftet. Die Wege von der Wurzel r bis zu den Blättern kodieren die Pattern P.
- Es findet nur eine Vorverarbeitung der Pattern statt. Es ist also möglich, ein und dieselbe Patternmenge gegen mehrere Texte zu matchen, ohne κ neu zu erstellen.
- Grundidee des Algorithmus ist eine Verallgemeinerung von Knuth-Morris-Pratt

Fehlerlinks: Pointer eines Knoten v aus κ zu einem Knoten n_v, der mit einem Suffix von $L(v)$ der Länge $lp(v)$ beschriftet ist. Für den Fall, dass $lp(v) = 0$ ist, zeigt dieser Pointer auf die Wurzel r.
Matching: Der Text T wird mit κ verglichen. Bei einem Mismatch von $T(c)$ und dem Zeichen der Kante (w, w') in κ wird ein Shift durchgeführt, so dass:

- $l = v - lp(w)$ mit $w = n_v$ falls $lp(w) > 0$
- oder $l = c, w = r$, falls $lp(w) = 0$

Anmerkungen:

- Die Fehlerlinks können in $O(n)$ bestimmt werden (Bws. durch vollst Induktion)
- Mit Hilfe von Outputlinks kann die Einschränkung umgangen werden, dass kein Pattern Substring eines anderen sein darf. (Sonst wird evtl. zu weit geshiftet !) Dabei sind Outputlinks Pointer von einem Knoten v zu dem nummerierten Knoten, der von v aus über die wenigsten Fehlerlinks erreichbar ist. Wir müssen also für jeden Knoten, den wir passieren überprüfen, ob er nummeriert ist, oder von ihm ein Pfad von Outputlinks ausgeht. Dadurch ist gewährleistet, dass keine Pattern verpasst werden.
- Die Outputlinks können während der Vorverarbeitung zusammen mit den Fehlerlinks in $O(n)$ bestimmt werden.

2.2 Anwendungen von Aho-Corasick

- Matching mit Wildcards in $0(km)$ (k = obere Schranke für die Anzahl der Wildcards)
- Zweidimensionales Matching in $O(n+m)$

3 Reguläre Ausdrücke und Automaten

Definition: Ein deterministischer, endlicher Automat (dfa) ist ein Tupel $(Q, \Sigma, \delta, s, F)$ wobei

- Q = endl. Menge von Zuständen
- Σ = endl. Alphabet
- $\delta : Q \times \Sigma \to Q$ ist die Übergangsfunktion
- s $\in$ Q = Startzustand
- F $\subset$ Q ist die Menge der akzept. Endzustände

Eine Sprache L heisst dfa-Sprache gdw. sie von einem dfa akzeptiert wird.
Satz (Kleene): Die regulären Sprachen sind genau die dfa-Sprachen.
Definition: Ein nfa ist ein Tupel (Q, Σ, s, F, δ)
dabei ist die Übergangsfunktion δ diesmal über die Potenzmenge von Q definiert:

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$$

Das macht eben das nichtdeterministische des nfa aus. Laut Adrian kann man sich das ungefähr so vorstellen, als ob der Automat sich gleichzeitig in mehreren Zuständen befindet. Ein nfa akzeptiert eine Sprache, wenn bei ihrer Abarbeitung ein Weg in einem der akzeptierten Zustände endet.
Wichtig: regüläre Sprache $\Leftrightarrow$ dfa-Sprache $\Leftrightarrow$ nfa-Sprache

3.1 Matching gegen reguläre Ausdrücke

Wir konstruieren zu dem regulären Ausdruck R einen nfa zum Ausdruck $\Sigma^* R$. Dann fassen wir die Zustände des Automaten zu Mengen $N(i)$ zusammen. Dabei ist:

- $N(0)$ = Der Startzustand s und alle Knoten, die von s aus durch ϵ-Übergange erreicht werden können.
- $N(i)$ = Alle Knoten v, die von einem Knoten aus $N(i-1)$ über eine Kante mit dem Label $T(i)$ gefolgt von 0 oder mehr ϵ-Übergangen erreicht werden können. (uff, so stehts aber im Gusfield)

Für jede Knotenmenge, die wir passieren, muss überprüft werden, ob sie einen gültigen Endzustand enthält. Falls der Text die Länge m und der reguläre Ausdruck n Operationen enthält, ist die Laufzeit dieses Verfahren $0(nm)$.

4 Seminumerische Verfahren (Gusfield S. 70)

4.1 Shift-And

Wir verwenden ein Matrix M mit $n * m$ Einträgen, die spaltenweise von links nach rechts ausgefüllt wird. Der Eintrag $M(i, j)$ ist 1 gdw die ersten i Buchstaben aus P den Substring aus T machten, der an der Stelle j endet und i Buchstaben lang ist. Für jeden Buchstaben x aus Σ gibt's einen Kontrollvektor U , der an der Stelle 1 ist, an der x im Text vorkommt.

- Erste Spalte initialisiert mit $M(1, 1) = 1$, falls $T(1) = P(1)$, Rest 0
- M_j = Shift(M_{j-1}) AND $U(T(J))$

Die Einträge $M(n, j)$ (die unterste Zeile) sind genau dann Eins, wenn an der Stelle j im Text ein Pattern endet. Der Algorithmus ist für kurze Pattern sehr effizient. Laufzeit: $O(nm)$

4.2 Karp-Rabin

Idee: Wir fassen das Pattern als Zahl im $|\Sigma|$-Zahlenraum auf. (Formulierung by Adrian)
Wir errechnen zu dem Pattern und allen Substrings aus T der Länge n Zahlen in Abhängigkeit vom Alphabet . Logischerweise haben wir genau dann ein Matching, wenn die Zahlen von Pattern und Substring gleich sind. Das Problem dabei ist, dass diese Zahlen evtl. sehr groß werden.(2er Potenz in der Formel). Deshalb dividieren wir beide Zahlen durch eine Primzahl p und merken uns den Rest (= Fingerprint). Damit wir nicht erst die Zahl berechnen müssen, und dann dividieren, wird das Hornerschema verwendet. Der Haken an der Sache ist, dass wir dann keine hundertprozentige W-keit mehr haben, dass bei gleichem Fingerprint ein Matching vorliegt. Wählt man p jedoch geschickt (s. Skript) kann man die W-keit für ein falsches Matching minimieren. Laufzeit: $O(m)$

5 Suffix-Bäume (Gusfield S.89)

Definition: Ein Suffix-Baum T für einen String S der Länge m ist ein gerichteter Baum mit genau m Blättern, durchnummeriert von 1 bis m. Er hat folgende Eigenschaften:

- Jeder interne Knoten hat den Ausgrad ≥ 2 und jede Kante ist mit einem Substring von S markiert.
- Keine zwei Kanten, die von einem Knoten ausgehen, sind mit dem gleichen Buchstaben beschriftet.
- Die Kantenmarkierungen von der Wurzel bis zum Blatt i ergeben das Suffix von S, welches an Position i beginnt.

Ein Suffixbaum kann naiv in $0(m^2)$ erstellt werden.
$\rightarrow$ Mit Ukkonen's Algorithmus geht das auch in $O(m)$.

5.1 Suffix-Baum Konstruktion nach Ukkonen

Der Algorithmus nach Ukkonen erstellt zuerst einen impliziten Suffixbaum in m Phasen. Jede Phase i ist wieder unterteilt in i Erweiterungen. Es gibt 3 Möglichkeiten, die Erweiterungen durchzuführen. In dieser Form lässt sich der Suffix-Baum in $O(m^2)$ konstruieren. Um die Laufzeit auf $O(m)$ zu drücken, sind drei Tricks nötig:

- Skip/Count: Suffixlinks folgen, Kantenlabel nicht explizit lesen, sondern nur 1. Buchstabe und Länge, dann zum nächsten Knoten springen
- Wenn das erste Mal in einer Phase der aktuelle Substring im Baum enthalten ist (Regel 3), nichts mehr tun.
- Kanten zu Blättern mit globalen Zähler markieren, Zähler zu Beginn jeder Phase um eins erhöhen (Regel 1 erledigt)

5.2 Anwendungen von Suffixbäumen

Mit einem Suffixbaum kann man lauter tolle Sachen machen, z.B.:

- Pattern-Matching in $O(|P| + k)$. (k = Anzahl der Vorkommen, nur Text wird vorverarbeitet)
- Datenbank von Pattern - finde für einen String S alle Strings aus der Datenbank, die S enthalten, in $O(n + k)$
- Längster gemeinsamer Teilstring in $O(|S_1| + |S_2|)$
- LCA = longest-common-ancestor-Anfragen in konstanter Zeit:
  1. Longest-Common-Extension in $O(n)$ (Vorverarbeitung)
  2. Maximale Palindrome
  3. Matchings mit $\leq k$ Fehlern
  4. k-common substring einer Gruppe von Strings

6 Dynamisches Programmieren

Dynamisches Programmieren ist ein wichtiges Entwurfsparadigma, wie Greedy-Strategien oder Divide-and-Conquer. Eine Voraussetzung für die Anwendbarkeit dieser Strategie ist, dass sich die optimale Lösung aus sich überlappenden optimalen Lösungen für Teilprobleme zusammensetzen lässt.

6.1 Editierabstand

Problem: Wir haben zwei Strings und interessieren uns für die minimale Anzahl der Operationen (Einfügen, Löschen und Ersetzen = Editierfolge), die notwendig sind, um den einen in den anderen String zu überführen. (= Editierabstand, Levenshtein-Distance)
Lösung: $D(i,j)$ ist definiert als die minimale Anzahl von Operationen, die notwendig ist, um die ersten i Buchstaben von S_1 in die ersten j Buchstaben von S_2 zu überführen. Wir versuchen das Problem $D(n,m)$ zu lösen, indem wir zuerst die Teilprobleme $D(i,j)$ für alle Kombinationen von i,j berechnen. Dies sind also die Teilprobleme, die sich überlappen, und deren optimale Lösungen kombiniert die optimale Lösung des Gesamtproblems ergibt.
general recurence:

$$\forall i : D(i,0) = i$$

$$\forall j : D(0,j) = j$$

$$D(i,j) = min\{D(i-1,j)+1, D(i,j-1)+1, D(i-1,j-1+t(i,j))\}$$

wobei $t(i,j) = 0$, wenn $S_1(i) = S_2(j)$ und 1 sonst.
Da diese rekursive Formel für die Berechnung eines Wertes immer auf vorherige Werte zugreift, speichern wir diese Werte in einer Tabelle. So muss kein Wert mehrfach berechnet werden. Wir speichern für jeden Eintrag der Tabelle einen oder mehrere Pointer auf die Einträge, die das Minimum zu Berechnung dieses Wertes speichern. Der Eintrag in der linken, unteren Ecke ergibt den Wert des Editierabstands. Folgen wir von dort den Pointern nach $(1,1)$, erhalten wir den/die optimale Editierfolge bzw. Alignment. (es kann mehrere geben!) Laufzeit: $O(nm)$

6.2 Optimales Alignment

Ein (globales) Alignment ist eigentlich nicht mehr als zwei Strings übereinander geschrieben, und zwar so, dass jeder Buchstabe entweder einem anderen Buchstaben oder einem Space gegenübersteht. (evtl. müssen am Anfang oder Ende Spaces eingefügt werden) Ein optimales Alignment ist ein Alignment, bei dem die Anzahl der Buchstaben, die mismatchen oder gegenüber Spaces stehen, minimal ist. Alignment und Editierfolge könne ineinander überführt werden, indem in der Editierfolge ein Einfügen als ein Space im oberen String, ein Löschen als Space im zweiten String und eine Ersetzung als zwei mismatchende Buchstaben übereinander interpretiert werden.

6.3 Hirschberg (Gusfield S. 254)

Das Problem an den Lösungen für das Alignment-Problem mit Dynamischer Programmierung ist der Speicherbedarf von $\Theta(nm)$ für die Tabelle. Der Hirschberg-Algorithmus benötigt nur $O(n)$ Speicher, während sich die Laufzeit verdoppelt (also linear bleibt). Dazu speichert der Algorithmus immer nur zwei Zeilen C und P der Tabelle. Die Grundidee des Algorithmus ist es, nicht top-down die

Ähnlichkeit der beiden Strings zu berechnen, sondern glechzeitig top-down und bottom-up bis zur Mitte der Tabelle.
Ablauf:

- Ertes Top- und Bottom-Problem lösen:

  $$V(n,m) = max_{0 \leq k \leq m}[V(n/2,k) + V^r(n/2+1, m-k)]$$

  In Worten heisst dies, das die Ähnlichkeit der beiden Strings gleich dem Maximum der aus n/2. Zeile von oben und der n/2+1. Zeile von unten ist. (stimmt nicht mit der Formel aus der Vorlesung überein, hat sich aber als richtig erwiesen).

- k^* = Position, welche das Maximum liefert. Dann geht ein Teilstück des optimalen Traceback-Pfades durch den Eintrag $(n/2, k^*)$ der Tabelle.

- k^* und dieses Teilstück können in $O(nm)$ Zeit und $O(m)$ Speicher bestimmt werden.

- Von hier aus starten wir einen Traceback vom Eintrag $V(n/2, k^*)$ aus der $n/2$. Zeile des Top- und Bottom-Problems.

- Das liefert uns einen Index k_1 in der $n/2-1$. Zeile und einen Index k_2 in der $n/2+1$. Zeile.

- Dies liefert uns ein neues Top- und ein neues Bottom-Problem. (Rekursion!)

Stichworte: Trade-Off (= Verbesserung des Speichers auf Kosten der Laufzeit), Ähnlichkeit von Strings, Traceback, hässliche Indexwurschtelei beim Programmieren

7 Turing-Maschinen

Definition: Eine deterministische Turingmaschine besteht aus

- Q endl. Zustandsmenge
- Σ endl. Eingabealphabet
- Γ endl. Arbeitsalphabet
- $s \in Q$ Startzustand
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times L, N, R$
- evtl: $F \subset Q$ akzeptierte Endzustände

Stichworte:

- Übergangsfunktion $\delta(q, a) = (p, b, l)$, lies: TM ist in Zustand q, liest Zeichen $a \in \Gamma$, überschreibt a mit b, geht einen Schritt nach links, neuer Zustand ist p.
- Eine Konfiguration ist eine Momentaufnahme vom linkesten Nicht-Blank bis zum rechtesten Nicht-Blank + Kopfposition + Zustand
- Es kann Eingaben geben, bei denen eine Turing-Maschine nicht stoppt.
- Eine Turingmaschine akzeptiert ein Wort, wenn sie bei seiner Abarbeitung in einem akzeptierten Zustand anhält.
- Die Klasse der Turing-berechenbaren Funktionen ist genau die Klasse der im intuitiven Sinne berechenbaren Fkt. *(Church-Turing)*