HUMBOLDT-UNIVERSITÄT ZU BERLIN

Exposé zur Studienarbeit

„Datenbankgestützte approximative Suche mit Präfix-Baum-Indizes“

Astrid Rheinländer

Betreuer: Prof. Dr. Ulf Leser

Motivation

Expressed Sequence Tags (ESTs) sind kurze, transkribierte Nukleotidsequenzen, die die exprimierten Gene einer Zelle beschreiben. ESTs können durch Ansequenzierung von cDNA-Klonen schnell und günstig generiert werden. Die Qualität von EST-Sequenzen ist häufig relativ schlecht, Gründe dafür sind u. a. die interindividuelle Varianz und Lesefehler bei der Sequenzierung. Die EST-Datenbank *dbEST* [5] enthält mittlerweile mehr als 55 Millionen identifizierte EST-Sequenzen. Eine Arbeit, insbes. das zielgerichtete Auffinden gespeicherter Informationen, auf diesen ungenauen Daten erfordert neben effizienten Zugriffsstrukturen auch die Möglichkeit approximativ zu suchen und zu joinen. Da kommerzielle RDBMS die Funktionalität des approximativen Joins nicht direkt unterstützen, muss dies durch *user-defined functions* (UDF) nachgerüstet werden.
Nicky Hochmuth hat in seiner Diplomarbeit den Präfix-Baum als Indexstruktur für EST-String-Attribute im kommerziellen RDBMS *Oracle Database* vorgestellt [3]. Darauf aufbauend wurde in der Diplomarbeit von Martin Knobloch Hochmuths Implementierung nach *C++* portiert und optimiert [4]. Knobloch konnte die Performanz des Präfix-Baums *CppPTree* in diversen Punkten sprunghaft verbessern, speziell die Ausführung des Joins auf *CppPTree*-indizierten Stringattributen konnte innerhalb von *Oracle* im Vergleich zu internen Operatoren schneller ausgeführt werden. In der vorhanden Implementierung steht nur die Suche mit Wildcards zur Verfügung, die Suche nach ESTs mit höchstens k Fehlern (k *Mismatch*) oder mit Levenshtein-Abstand (*eLA*) $\leq k$ fehlt. Ebenso ist ein approximativer Join mit k *Mismatches* oder *eLA* bisher nicht möglich.

Zielstellung

Ziel dieser Arbeit ist, Funktionalität für die approximative Suche und den approximativen Join in *CppPTree* bereitzustellen. Es sollen folgende Varianten implementiert werden:

1. Für ein Pattern P, eine Zahl k und eine EST-Menge $S = \{S_1, ..., S_n\}$:
   k-Mismatch: Finde alle $S_i \in S, |S_i| = |P|$, die sich von P in höchstens k Stellen unterscheiden.
   eLA: Finde alle $S_i \in S$, die einen *eLA* von höchstens k zu P haben.

2. Für zwei EST-Mengen $P = \{P_1,...,P_m\}$, $S = \{S_1,...,S_n\}$ und eine Zahl k:
   k-Mismatch: Für alle $P_i \in P$, finde alle $S_j \in S, |S_j| = |P_i|$, die sich von P_i in höchstens k Stellen unterscheiden.
   eLA: Für alle $P_i \in P$, finde alle $S_j \in S$ mit *eLA* höchstens k zu P_i.

Die Effizienz der approximativen Suche soll anhand beispielhafter Daten aus *dbEST* [5] verifiziert werden. Darüber hinaus soll die vervollständigte *CppPTree*-Indexstruktur für die Integration als UDF in das kommerzielle RDBMS *Oracle Database* angepasst werden.

Techniken

1. ***eLA* als Ähnlichkeitsmaß für zwei Strings**
   Der einfache Levenshtein-Abstand (*eLA*) für zwei Strings *P, S* bezeichnet die geringste Anzahl benötigter Insertions, Deletions und Replacements für die Transformation von *P* in *S* und kann algorithmisch mit Hilfe der dynamischen Programmierung berechnet werden. Die Ähnlichkeit von *P* und *S* wird auf die Ähnlichkeit zweier Teilstrings von *P* und *S* reduziert, sukzessive wird der *eLA* für wachsende Präfixe von *P* und *S* berechnet. Dabei sind alle folgenden Teillösungen, und damit auch die Gesamtlösung, mindestens so groß wie die aktuell kleinste Teillösung. Außerdem ist die Gesamtlösung mindestens so groß wie die Differenz der Längen von P und S. Die Zeit- und Platzkomplexität des Verfahrens beträgt für zwei Strings jeweils $O(|P|*|S|)$.

2. **Approximatives Suchen in Präfix-Bäumen**
   Shang/Merrett haben eine Methode zur approximativen Suche in Präfixbäumen vorgestellt, die auf *Depth-First*-Traversierung und gleichzeitiger dynamischer Programmierung basiert [7]. Dabei werden folgende Eigenschaften des Präfixbaums und der dynamischen Programmierung ausgenutzt:

   (1) Sei x ein Knoten im Präfixbaum T, der das Präfix $S[1..x]$ repräsentiert und P ein Pattern. Dann haben alle Strings $S_1,\ldots,S_n$ im Teilbaum unterhalb von x das Präfix $S[1..x]$ gemeinsam, die Zeilen 0 bis x der Distanzmatrix $M[S_i,P]$ sind für alle S_i identisch. Dieser Bereich muss deshalb nur einmal ausgewertet werden und somit wird im Average Case der Aufwand für die Berechnung des *eLA* reduziert.

   (2) Falls in einer Zeile j, $0 \le j \le x$, von M ausschließlich Werte $> k$ enthalten sind, kann kein String S_i mit dem Präfix $S[1..x]$ einen $eLA(P, S_i) \le k$ erreichen. Deshalb kann der Teilbaum ab Knoten x für die weitere Suche ignoriert werden.

   Die Suche nach $eLA(P, S) \le k$ für alle Strings S in T hat die Komplexität $O(|P|*|T|*s)$, wobei s die Maximallänge der Strings in T bezeichnet. Shang/Merrett haben dieses Verfahren auf englischem Text und auf Quellcodes von C-Programmen ausgewertet und mit der Performanz von agrep verglichen. Für $0 \le k \le 1$ war die Suche im Präfixbaum performant, für wachsende k war agrep durchweg schneller.
   Hochmuth hat das Verfahren für präfix-baum-indizierte ESTs implementiert und experimentell mit der Performanz einer sortierten Stringliste verglichen [3]. Im Ergebnis benötigte die Suche in der Stringliste durchschnittlich die Hälfte der Rechenzeit der Suche im Präfixbaum. Hochmuth führt dies auf den Ausschluss von Strings aufgrund ihrer Länge in der Stringliste zurück.

Für die Suche nach *k-Mismatches* ist keine dynamische Programmierung erforderlich, da in diesem Fall nur Strings mit gleicher Länge zum Pattern gefragt sind, die beliebige k Fehler enthalten dürfen. Eigenschaften (1) und (2) gelten analog, da die Fehlerzahl für gemeinsame Präfixe nur einmal ermittelt werden muss und die Anzahl der Fehler für wachsende Präfixe nie kleiner als die bisherige Fehlerzahl wird.

3. Einschränken des Suchraums durch Filter

Aufgrund der Komplexität der *eLA*-Berechnung sollten Filtermethoden eingesetzt werden, die frühzeitig möglichst große Teilbäume aus dem Suchraum ausschließen und so die Kosten reduzieren.

<u>Längenfilter</u>

Zwei Strings S, P können nur dann $eLA(S,P) \leq k$ haben, falls $|S-P| \leq k$ gilt. Im Präfixbaum sind die exakten Stringlängen jedoch nur an den Knoten zu ermitteln, die Strings repräsentieren. Für alle anderen Knoten ist die Länge des Präfix bekannt, die Länge aller Suffixe im darunter liegenden Teilbaum nicht. Eine Variante wäre, für jeden Knoten eine Liste der Längen der Strings im Subbaum zu verwalten. Dies hätte einen quadratischen Aufwand von $O(|Anzahl\ der\ Knoten\ in\ T| * |Anzahl\ der\ Strings\ in\ T|)$ zur Folge und ist somit nicht performant.

Alternativ kann für jeden Knoten x die Länge des kürzesten (*minLength*) und längsten (*maxLength*) Strings im Teilbaum unterhalb von x gespeichert werden. Falls

$$|x.maxLength + k| < P.Length \text{ oder } |x.minLength - k| > P.Length$$

gilt, enthält der Teilbaum unterhalb von x keine Treffer mit $eLA(S,P) \leq k$ bzw. maximal k Fehlern und kann daher komplett ignoriert werden. Die Ermittlung von *maxLength* und *minLength* kann während der Indexerstellung mit konstantem Aufwand erfolgen. Pro Knoten entsteht durch die zwei zusätzlichen Attribute ein Overhead von O(2).

Für Knoten x in Präfixbaum T_1 und Knoten y in Präfixbaum T_2 sind die Längen des jeweils kürzesten und des längsten Strings im Teilbaum unterhalb von x und y bekannt. Deshalb gilt: Die Teilbäume ab x und ab y enthalten potentiell Strings $S_i \in T_1(x), S_j \in T_2(y)$ mit $eLA(S_i, S_j) \leq k$ bzw. maximal k Fehlern, wenn

$$|y.minLength - x.maxLength| \leq k \text{ oder } |x.minLength - y.maxLength| \leq k$$

gilt.

<u>Q-Gramm-Filter [2]</u>

Der Suchraum kann auch durch Ausnutzung der Eigenschaften von q-Grammen eingeschränkt werden, da zwei Strings S und P mit relativ kurzem *eLA* zueinander auch eine große Anzahl q-Gramme gemeinsam haben. Dabei gilt für die Strings und die Mengen der korrespondierenden q-Gramme G_S, G_P und $eLA(S,P) \leq k$:

$$eLA(S,P) \leq k \Leftrightarrow G_S \cap G_P \geq \max(|S|,|P|) - 1 - (k-1) * q.$$

Gravano et al. [2] konnten mit Hilfe des q-Gramm-Filters und eines Längenfilters den Suchraum für approximative Joins auf Telefonbuchdaten enorm einschränken. Die Anzahl der Join-Kandidaten, die schließlich mit dynamischer Programmierung untersucht werden mussten, verringerte sich um 99%, die Ausführung des Joins konnte um das 20-fache im Vergleich zur Variante ohne Filter gesteigert werden. Die Berechnung der q-Gramme erfolgte nicht im Voraus, sondern als Teil des Preprocessings während des Joins. Der zusätzliche Aufwand für die q-Gramm-Berechnung im Vergleich mit der Ausführungszeit des Joins war vernachlässigbar.

Dieser Ansatz soll auf den CppPTree übertragen werden. Die q-Gramme G_x für das Präfix $P[1..x]$ an Knoten x werden erst zu dem Zeitpunkt berechnet, an dem x das erste Mal besucht wird. Da die Menge der q-Gramme $G_{x'}$ für den Vaterknoten x' von x bereits bekannt ist, kann G_x aus $G_{x'}$ berechnet werden:

$$G_x = G_{x'} - \{z \mid z \text{ ist } q\text{-Gramm in } P[(x'-q+1)..x']\} \cup \{y \mid y \text{ ist } q\text{-Gramm in } P[(x'-q+1)..x]\}.$$

Die Wahl von q sollte im Bereich von $\log_{|\Sigma|}(|S|)$ liegen [5], für ESTs auf einem 4-buchstabigen Alphabet ist dies das Intervall [3,6].

<u>Häufigkeitsvektoren [1]</u>

Für einen String S über einem Alphabet $\Sigma = \alpha_1,...,\alpha_l$ enthält der Häufigkeitsvektor $f(S) = [f_1,...,f_l]$ in jeder Komponente die Anzahl der Vorkommen von Zeichen α_i in S. Der Häufigkeitsabstand *fDist(f(S),f(P))* für zwei Strings *S, P* ist die kleinste Anzahl $+1, -1, \pm 1$-Operationen, um den *f(S)* in *f(P)* zu transformieren. Aghili et al. [1] haben gezeigt, dass *fDist* eine untere Schranke für den *eLA* darstellt:

$$fDist(f(S), f(P)) > k \Rightarrow eLA(S,P) > k.$$

Der Häufigkeitsabstand liefert also ein hinreichendes Kriterium, Strings frühzeitig aus der Menge der potentiellen Treffer auszuschließen.

Der Häufigkeitsvektor kann für jedes Präfix bereits während der Indexerstellung berechnet werden. Über einem 4-buchstabigen Alphabet ergibt sich so ein Overhead von 4 Integer-Werten pro Knoten.

Arbeitsschritte

1. **Attributerweiterung der Knoten in *CppPTree***

   Für jedes Knotenobjekt x sollen zusätzlich folgende Attribute gespeichert werden:
   - *minLength:* Länge des kürzesten Strings, der sich im Teilbaum unterhalb von x befindet,
   - *maxLength:* Länge des längsten Strings, der sich im Teilbaum unterhalb von x befindet,
   - Häufigkeitsvektor *f[S(1..x)],* der die Anzahl der a,c,g,t im aktuellen Präfix *S(1..x)* enthält.

2. **Implementierung der Suche nach *k-Mismatches***

   Der Präfixbaum wird *Depth-First* traversiert, an jedem Knoten x wird der Längenfilter in Bezug auf die Patternlänge eingesetzt. Der Häufigkeitsfilter kann auch an jedem Knoten angewandt werden, da für *k-Mismatches* nur Strings S mit der gleichen Länge des Patterns P gesucht werden. Deshalb gilt für Präfixe gleicher Länge x von S und P:

   $$fDist(f(S(1..x), f(P(1..x)) > k \Rightarrow eLA(S(1..x), P(1..x)) > k.$$

   Siehe auch Anhang, Algorithmus 1.

   Für *k-Mismatches* auf zwei Präfixbäumen erfolgt die Suche analog, indem T_1 und T_2 gleichzeitig traversiert werden und jeweils für wachsende Präfixe der Strings in T_1 und T_2 die Anzahl der Fehler ermittelt wird.

3. **Implementierung der Suche nach *eLA* ≤ *k***

   Für die Suche nach einem String P im Präfix-Baum T wird T *Depth-First* traversiert. An jedem Knoten wird der Längenfilter und/oder der q-Gramm-Filter eingesetzt. Falls einer dieser Filter greift, wird der Teilbaum ab x nicht weiter betrachtet. Sonst wird die Distanzmatrix $M[S_x,P]$ berechnet, falls $M[S_x,P]$ ausschließlich Werte $> k$ enthält, wird der Teilbaum ab x nicht weiter untersucht (siehe Anhang, Algorithmus 2).

Für *eLA* auf zwei Präfixbäumen erfolgt die Suche analog, indem T_1 und T_2 gleichzeitig traversiert werden und der *eLA* für jeweils wachsende Präfixe der Strings in T_1 und T_2 berechnet wird.
Der Häufigkeitsfilter erscheint für den Vergleich von zwei echten Präfixen in inneren Knoten zunächst nicht praktikabel, für die Verarbeitung von Suffixen ergibt sich durchaus ein Einsatzfeld. Suffix-Knoten sind stets Blätter des Präfix-Baums, die eigentlichen Suffixe sind in einer separaten Suffix-Datei auf der Festplatte abgelegt. Der Gedanke ist, Zugriffe auf diese Datei solange wie möglich zu vermeiden: Angenommen, Pattern *P* wird untersucht und der Baumabstieg ist bis zu einem Blatt in *T* fortgeschritten, das String *S* repräsentiert. *fDist(f(P),f(S))* wird ermittelt und das Suffix von *S* wird nur dann von der Platte geladen und mit dynamischer Programmierung untersucht, falls $fDist(f(S), f(P)) \leq k$ gilt.

4. Integration in *Oracle Database* als UDF

Für die Nutzung in Oracle Database muss *CppPTree* am *Oracle Data Cartridge Interface* (ODCI) in Form einer shared library angemeldet werden. Für Knoblochs Implementierung liegt die Anbindung bereits vor und muss nun um die neuen Funktionen erweitert werden. Für Details zur Verfahrensweise siehe [4].

5. Performance-Analyse

Die Effektivität der Filterstrategien Länge, *q*-Gramme und Häufigkeitsvektoren soll einzeln und in verschiedenen Kombinationen untersucht werden. Zum Vergleich werden Algorithmen implementiert, die auf jegliche Vorfilterung verzichten. Experimentell wird eine optimale Länge der *q*-Gramme für ESTs ermittelt.

Literatur

[1] AGHILI, S. A.; AGRAWAL D.; AND ABBADI, A. E. *BFT: Bit Filtration Technique for Approximate String Join in Biological Databases. Proceedings of the 10th Symposium on String Processing and Information Retrieval (SPIRE 2003).*

[2] GRAVANO, L.; IPEIROTIS, P.; JAGADISH, H. V.; KOUDAS, N.; MUTHUKRISHNAN, S.; AND SRIVASTAVA, D. *Approximate String Joins in a Database (Almost) for Free. Proceedings of the 27th VLDB Conference, Roma, Italy, 2001.*

[3] HOCHMUTH, N. *Präfix-Bäume als Indexstruktur für String-Attribute in relationalen Datenbanken. Diplomarbeit, Humboldt-Universität zu Berlin, Berlin, 2006.*

[4] KNOBLOCH, M. *Optimierung von Präfix-Baum-Indizes für String-Attribute. Diplomarbeit, Humboldt-Universität zu Berlin, Berlin, 2007.*

[5] NATIONAL CENTER FOR BIOTECHNOLOGY INFORMATION. *Expressed Sequence Tags database dbESt. URL,* http://www.ncbi.nlm.nih.gov/dbEST/.

[6] NAVARRO, G. *A guided toir to approximate string matching. ACM Computing Surveys, Vol. 33, 2001, S. 31-88.*

[7] SHANG, H.; AND MERRETT, T.H. *Tries for Approximate String Matching. IEEE Transactions on Knowledge and Data Engineering, Vol. 8, 1996, S.540-547.*

Anhang

Algorithmus 1: Suche nach allen Vorkommen eines Patterns im Präfixbaum mit höchstens k Fehlern

```
kMismatchSearch (Pattern P, CppPTree T, k) {

  Node x = T.root
  p = 0                                                    /* P[1..p] was already checked */
  faults = 0                                               /* no. of seen mismatches */

  While T contains unmarked nodes {
      Mark x as visited

      If (x.maxLength + k < P.length) or (x.minLength − k > P.length) {   /* apply length filtering */
            Mark all children of x as visited
            x = x.nextElement                              /* go on with next item
            continue                                          in dfs-order */
      }

      Compute frequency-vector f(P[1..x])
      If frequency-distance of f(S[1..x]) and f(P[1..x]) > k {   /* apply frequency filtering * /
            Mark all children of x as visited
            x = x.nextElement                              /* go on with next item
            continue                                          in dfs-order */
      }

      faults = faults+ no. of mismatches in S[p+1..x] and P[p+1..x]   /* compare iff both filters
      If (faults > k) {                                      were passed */
            Mark all children of x as visited
            x = x.nextElement
            continue
      }

      p=x                                                  /* update last checked position */

      If x contains String-ID and p = P.length {
                output String-ID                           /* a match occurred */
      }
      x = x.nextElement                                    /* go on with next item in
  }                                                           dfs-order */
}
```

Algorithmus 2: Suche nach allen Vorkommen eines Patterns im Präfixbaum mit eLA höchstens k

```
eLASearch (Pattern P, CppPTree T, k) {

  Node x = T.root

  While T contains unmarked nodes {
      Mark x as visited
      If (x.maxLength + k < P.length) or (x.minLength − k > P.length) {        /* apply length filtering */
            Mark all children of x as visited
            x = x.nextElement                                                   /* go on with next item
            continue                                                               in dfs-order */
      }

      G_x = set of q-grams in prefix S[1..x]
      y = min(P.Length, x)                                                      /* P will be evaluated until y */
      G_P = set of q-grams in P[1..y]

      If G_x ∩ G_P < max(|S_x|, y) − 1 − (k − 1) {                              /* apply q-gram filtering */
            Mark all children of x as visited
            x = x.nextElement                                                   /* go on with next item
            continue                                                               in dfs-order */
      }

      If x is leaf {                                                            /* apply frequency filtering */
            Compute frequency-vector f(P)
            If frequency-distance of f(S) and f(P) > k {
            x = x.nextElement                                                   /* go on with next item
            continue                                                               in dfs-order */
      }

      Compute distance-matrix M with eLA(S_x, P)                                /* dynamic programming */
      If M contains a value M[i, j] ≤ k {
            if x contains String-ID {                                           /* possible match */
                  If M[n, m] ≤ k M[n,m] output String-ID
            }

      }
      Else {                                                                    /* eLA>k*/
            Mark all children of x as visited
      }
      x=x.nextElement                                                           /* go on with next item
  }                                                                                in dfs-order */
}
```