Lehrstuhl für Informatik I
Prof. Juraj Hromkovič
Dr. Hans-J. Böckenhauer
Dirk Bongartz

1. Übung zur Vorlesung Algorithmische Grundlagen der Bioinformatik

Aachen, den 30. April 2003

Aufgabe 1

(a) Bestimmen Sie die maximale Anzahl verschiedener Teilsequenzen in einem String der Länge n.

(b) Seien $t = t_1 \dots t_n$ und $p = p_1 \dots p_m$ zwei Strings über einem Alphabet Σ. Geben Sie einen effizienten Algorithmus an, der entscheidet, ob p eine Teilsequenz von t ist, und analysieren Sie die Laufzeit Ihres Algorithmus.

10 Punkte

Aufgabe 2

Sei $S = \{s_1, \dots, s_k\}$ eine Menge von Strings über einem Alphabet Σ. Geben Sie einen Algorithmus an, der alle paarweisen Overlaps der Strings aus S bestimmt, und analysieren Sie die Laufzeit Ihres Algorithmus. Für die Laufzeitanalyse dürfen Sie davon ausgehen, dass alle Strings in S dieselbe Länge haben. **10 Punkte**

Aufgabe 3

Sei $t = t_1 \dots t_n$ ein String über einem Alphabet Σ. Ein *zyklischer Shift* von t ist ein String $t_i \dots t_n t_1 \dots t_{i-1}$ für ein $i \in \{1, \dots, n\}$. Sei $p = p_1 \dots p_m$ ein weiterer String über Σ.

(a) Geben Sie einen möglichst effizienten Algorithmus an, der überprüft, ob p ein zyklischer Shift von t ist, und analysieren Sie dessen Laufzeit.

(b) Geben Sie einen möglichst effizienten Algorithmus an, der überprüft, ob p ein Teilstring eines zyklischen Shifts von t ist, und analysieren Sie dessen Laufzeit.

10 Punkte

Aufgabe 4

Sei $t = t_1 \dots t_n$ ein String über einem Alphabet Σ, sei $p = p_1 \dots, p_m$ ein String über $\Sigma \cup \{\star\}$. Dabei bezeichne $\star$ ein Lückensymbol, das für beliebig viele Symbole aus Σ stehen kann. Wir sagen, dass p in t als Muster vorkommt, falls es für jedes $j \in \{1, \dots, m\}$ einen String $s_j \in \Sigma^*$ gibt mit $s_j = p_j$ für alle $p_j \neq \star$, so dass $s_1 \cdot \ldots \cdot s_m$ ein Teilstring von t ist.
Zum Beispiel kommt das Muster $p = ab \star ab \star c$ in dem Text $t = abccabca$ vor:

$$\underbrace{ab}_{ab}\underbrace{cc}_{\star}\underbrace{ab}_{ab}\underbrace{\quad}_{\star}\underbrace{c}_{c}a$$

Geben Sie einen möglichst effizienten Algorithmus an, der für einen gegebenen Text $t \in \Sigma^*$ und ein gegebenes Muster p über $\Sigma \cup \{\star\}$ mit Lückensymbolen entscheidet, ob p in t vorkommt. Analysieren Sie die Laufzeit Ihres Algorithmus. **10 Punkte**

Allgemeine Hinweise:

- Geben Sie Ihre Algorithmen in Pseudocode-Notation an und erläutern Sie die zugrunde liegende Idee bzw. begründen Sie informell die Korrektheit der Algorithmen.
- Bei der Laufzeitanalyse der Algorithmen reicht es aus, eine (möglichst gute) obere Schranke auf die Laufzeit in der O-Notation zu bestimmen.

Abgabe: bis Mittwoch, den 7. Mai 2003, um 14.00 Uhr im Sammelkasten am Lehrstuhl.

Lehrstuhl für Informatik I
Prof. Juraj Hromkovič
Dr. Hans-J. Böckenhauer
Dirk Bongartz

Musterlösung zur 1. Übung zur Vorlesung Algorithmische Grundlagen der Bioinformatik

Aachen, den 12. Mai 2003

Lösung zu Aufgabe 1

(a) Betrachten wir einen String $t = t_1\, t_2 \ldots t_n$ der Länge n. Die maximale Anzahl *verschiedener* Teilsequenzen haben wir genau dann, wenn keine zwei Buchstaben im String t gleich sind. Dann entspricht jede Teilmenge von Positionen in t genau einer möglichen Teilsequenz, also gibt es in diesem Fall 2^n verschiedene Teilsequenzen (darunter auch die leere Teilsequenz).

(b) Ein effizienter Algorithmus, der entscheidet, ob ein Muster $p = p_1 \ldots p_m$ eine Teilsequenz des Textes $t = t_1 \ldots t_n$ ist, kann wie folgt arbeiten. Die in Klammern eingeschlossenen Kommentare dienen der Erläuterung.

Eingabe: Ein Text $t = t_1 \ldots t_n$ und ein Muster $p = p_1 \ldots p_m$
if $m > n$ **then** {Wenn das Muster länger ist als der Text, dann kann es keine Teilsequenz des Textes sein.}
return „NEIN“
$j := 1$ {durchläuft den Text, d.h. $1 \leq j \leq n$.}
$k := 1$ {durchläuft das Muster, d.h. $1 \leq k \leq m$.}
for $i := j$ **to** n **do**
if $p_k = t_i$ **then** {Aktuelles Symbol des Musters wurde im Text gefunden}
if $k = n$ **then** {p ist eine Teilsequenz von t.}
return „JA“
else
$k := k + 1$ {Suche nach dem nächsten Symbol des Musters ...}
$j := i + 1$ {...und zwar im verbleibenden Text}
else
return „NEIN“ {Das Muster ist nicht im Text enthalten.}

Es ist offensichtlich, dass der Algorithmus genau das Geforderte leistet. Da der Algorithmus sowohl den Text t als auch das Muster p jeweils höchstens einmal durchläuft, besitzt der Algorithmus eine Laufzeit in $O(n + m)$.

Lösung zu Aufgabe 2

Betrachte zunächst zwei Strings $s = s_1 \dots s_m$ und $t = t_1 \dots t_n$. Dann berechnet sich der maximale Overlap $Ov(s,t)$ dieser beiden Strings wie folgt.

Eingabe: Zwei Strings $s = s_1 \dots s_m$ und $t = t_1 \dots t_n$.

```
if m ≤ n then
    k := 2 {Maximal mögliche Startposition des Overlaps in s, falls |s| ≤ |t|.}
else {m > n}
    k := m − n + 2 {Maximal mögliche Startposition des Overlaps in s, falls |s| > |t|.}
Ov(s,t) := λ {Initialisierung des Overlaps}
found := false {Overlap noch nicht gefunden}
while found = false and k ≤ m do
    i := k
    j := 1
    while s_i = t_j and i ≤ m do
        i := i + 1
        j := j + 1
    if i > m then
        Ov(s,t) := s_k ... s_m
        found := true
    else
        k := k + 1
```

Ausgabe: Den Overlap $Ov(s,t)$ von s und t.

Offenbar berechnet dieser Algorithmus den korrekten Overlap von s und t. Der Algorithmus erreicht dann die maximale Laufzeit, wenn der Overlap von s und t leer ist. Falls s und t dieselbe Länge n haben, benötigt der Algorithmus in diesem Fall eine Laufzeit in $O(n^2)$, da für jede mögliche Startposition $k \in \{2, \dots, n\}$ jeweils $s_k \dots s_n$ mit dem Präfix von t verglichen wird.

Um die Menge aller paarweisen Overlaps der Strings aus $S = \{s_1, \dots, s_k\}$ zu bestimmen, führt man den oben aufgeführten Algorithmus für alle $O(k^2)$ möglichen Paare von Strings aus $S = \{s_1, \dots, s_k\}$ durch und erhält so als Gesamtkomplexität $O(n^2 \cdot k^2)$.

Lösung zu Aufgabe 3

(a) Eine Idee für die Lösung ist die folgende. Wir bilden die Konkatenation $t' = t \cdot t = t_1 \dots t_n \, t_1 \dots t_n$ von t mit sich selbst. Es ist leicht einzusehen, dass der Text t' alle möglichen zyklischen Shifts von t enthält. Um zu überprüfen, ob ein gegebener String p ein zyklischer Shift von t ist, sind zwei Schritte erforderlich.

1. Prüfe zunächst, ob $n = m$ gilt, d.h. ob $|p| = |t|$. Falls nein, kann das Muster p kein zyklischer Shift des Textes t sein. Gebe die Antwort „NEIN“ aus. Diese Überprüfung kann in $O(|t'|) = O(n)$ durchgeführt werden.
2. Falls $n = m$ gilt, d.h. Text und Muster gleich lang sind, wende einen der in der Vorlesung besprochenen String-Matching-Algorithmen auf t' und p an. Dieses String-Matching kann beispielsweise mit Hilfe eines String-Matching-Automaten in

$$O(|t'| + |\Sigma| \cdot |p|) = O((|\Sigma| + 2) \cdot n) = O(|\Sigma| \cdot n)$$

erfolgen (siehe Laufzeitbetrachtung zu Algorithmus 4.3 der Vorlesung).

Die Gesamtlaufzeit des Algorithmus, die sich aus den Schritten 1 und 2 ergibt, beläuft sich schließlich auf $O(n + |\Sigma| \cdot n) = O(|\Sigma| \cdot n)$.

(b) Um zu überprüfen, ob p ein Teilstring eines zyklischen Shifts von t ist, ersetze im Algorithmus aus Teilaufgabe (a) die Prüfung auf $|p| = |t|$ durch die Prüfung auf $|p| \leq |t|$, d.h.

breche den Algorithmus in Schritt 1 dann ab, wenn $|p| > |t|$ gilt. Eine analoge Laufzeitanalyse wie in (a) ergibt auch hier eine Laufzeit in $O(|\Sigma| \cdot n)$.

Lösung zu Aufgabe 4

Die Idee für einen Algorithmus, der überprüft, ob der String p im Text t als Muster vorkommt, ist die folgende.

1. Falls das Muster p kein Sternsymbol $\star$ enthält, führe ein einfaches String-Matching von p in t durch. Wenn p in t vollständig enthalten ist, beende den Algorithmus mit Ausgabe „JA“, ansonsten gebe „NEIN“ aus.

2. Sonst lese das Präfix p' von p bis zum ersten Auftreten eines Sternsymbols, d.h. $p = p' \star p'', p' \in \Sigma^*, p'' \in (\Sigma \cup \{\star\})^*$. Führe ein String-Matching mit p' und t durch. Ist p' vollständig in t enthalten, z.B. $t = t' \cdot p' \cdot t''$, wiederhole die Schritte 1 und 2 mit $p := p''$ und $t := t''$. Ansonsten gebe die Antwort „NEIN“ aus.

Dieser Algorithmus lässt sich auch wie folgt darstellen:

Eingabe: Ein Text $t = t_1 \dots t_n \in \Sigma^n$ und ein Muster $p = p_1 \dots p_m \in (\Sigma \cup \{\star\})^m$.
Finde alle $\star$-Symbole in p, sei $p = q_1 \star q_2 \star \dots \star q_k$ für $q_1, \dots, q_k \in \Sigma^*$.
$t' := t$
for $i := 1$ **to** k **do**
Finde erstes Vorkommen von q_i in t' mit beliebigem String-Matching-Verfahren, falls vorhanden.
if q_i ist kein Teilstring von t' **then**
return „NEIN“
else
Sei $t' = t'' q_i t'''$ für das erste Vorkommen von q_i in t'.
$t' := t'''$
return „JA“

Die Laufzeit des Algorithmus wird im Wesentlichen durch das String-Matching der Fragmente $q_1, \dots, q_k$ des Musters p mit dem Text t bestimmt. Wir bezeichnen die maximale Länge eines solchen Fragments, d. h. eines Teilstrings $q_i \in \Sigma^*$ von p, der lediglich aus Buchstaben in Σ besteht, mit s. Die Konstruktion der String-Matching-Automaten für alle diese k Fragmente kann in Zeit $O(k \cdot |\Sigma| \cdot s)$ durchgeführt werden. Die Bestimmung dieser Fragmente ist offenbar in einer Zeit in $O(m)$ möglich. Eine obere Schranke für die Gesamtlaufzeit des Algorithmus ist dann $O(m + n + k \cdot |\Sigma| \cdot s)$, wobei m die Länge des Musters und n die Länge des Textes t bezeichnet.

Lehrstuhl für Informatik I
Prof. Juraj Hromkovič
Dr. Hans-J. Böckenhauer
Dirk Bongartz

2. Übung zur Vorlesung Algorithmische Grundlagen der Bioinformatik

Aachen, den 14. Mai 2003

Aufgabe 5

Geben Sie einen Text t der Länge n und ein Muster p der Länge m an, so dass der Boyer-Moore-Algorithmus das String-Matching-Problem für p und t in einer Zeit in $\Theta(\frac{n}{m})$ löst, wohingegen die alleinige Anwendung der Good-Suffix-Regel zu einer Laufzeit in $\Theta(n)$ führt. **10 Punkte**

Aufgabe 6

Sei t ein String der Länge n über einem Alphabet Σ, sei $\mathcal{S}$ eine Multimenge von m Symbolen aus Σ. Geben Sie einen Algorithmus an, der alle Teilstrings der Länge m in t findet, die genau die Symbole aus $\mathcal{S}$ enthalten. Ihr Algorithmus sollte mit einer Laufzeit in $O(n \cdot |\Sigma|)$ auskommen.

Motivation: Dieser Algorithmus kann für einen Ansatz zur Sequenzierung von Proteinen verwendet werden: Die experimentelle Sequenzierung von Proteinen ist sehr aufwändig, man kann aber für ein gegebenes Protein relativ einfach die Multimenge der enthaltenen Aminosäuren bestimmen. Wenn man nun das vollständige Genom schon sequenziert hat und annimmt, dass das Gen, welches das gegebene Protein kodiert, ein zusammenhängendes Teilstück des Genoms darstellt (wie es bei Prokaryonten häufig der Fall ist), dann kann man die Basensequenz des Genoms in eine Aminosäuresequenz übersetzen und mit dem obigen Algorithmus die Kandidaten für die Sequenz des Proteins bestimmen. **10 Punkte**

Aufgabe 7

Sei $t = t_1 \dots t_n$ ein String. Für alle $i \in \{1, \dots, n\}$ sei $Prior(i)$ das längste Präfix von $t_i \dots t_n$, das auch ein Teilstring von $t_1 \dots t_{i-1}$ ist. Weiter sei $l_i = |Prior(i)|$ und s_i die Anfangsposition des am weitesten links stehenden Vorkommens von $Prior(i)$ in t.

Eine Variante des sogenannten Ziv-Lempel-Verfahrens verwendet die Werte l_i und s_i zur komprimierten Darstellung von t. Dieses Verfahren beruht auf der folgenden Idee: Wenn man schon eine komprimierte Darstellung von $t_1 \dots t_{i-1}$ gegeben hat, und $Prior(i) \neq \lambda$ gilt, dann kann man die nächsten l_i Zeichen von t eindeutig durch (s_i, l_i) beschreiben. Damit ergibt sich zum Beispiel für den String $t = ababababababababababababababab$ die komprimierte Darstellung $ZL(t) = ab(1,2)(1,4)(1,8)$.

Geben Sie einen Algorithmus an, der mit Hilfe eines Suffix-Baums für t die komprimierte Darstellung $ZL(t)$ berechnet. Ihr Algorithmus sollte mit einer Laufzeit in $O(n \cdot \log n)$ auskommen. **10 Punkte**

(bitte wenden)

Aufgabe 8

Sei Σ ein Alphabet, auf dem eine lineare Ordnung $\prec$ gegeben sei. Dann lässt sich auf Σ^* die *lexikographische Ordnung* wie folgt definieren: Seien $s, t \in \Sigma^*$. Dann gilt $s \prec t$ genau dann, wenn s ein echtes Präfix von t ist, oder s und t sich zerlegen lassen als $s = uas'$ und $t = ubt'$ mit $u, s', t' \in \Sigma^*$, $a, b \in \Sigma$ und $a \prec b$.

Sei t ein String der Länge n über Σ. Wir wollen im Folgenden eine Datenstruktur betrachten, die platzsparender ist als ein Suffix-Baum, mit deren Hilfe sich das Teilstring-Problem aber immer noch effizient lösen lässt.

Ein *Suffix-Array* für t ist ein Array der Länge n, das die Zahlen $1, \ldots, n$ derart enthält, dass dadurch die lexikographische Ordnung aller (nichtleeren) Suffixe von t gegeben ist.

(a) Geben Sie einen effizienten Algorithmus an, der aus einem einfachen Suffixbaum für t das Suffix-Array für t konstruiert.

(b) Geben Sie einen Algorithmus an, der das Suffix-Array für t dafür verwendet, um für ein Muster p der Länge m in Zeit $O(m \cdot \log n)$ zu entscheiden, ob p ein Teilstring von t ist.

10 Punkte

Abgabe: bis Mittwoch, den 21. Mai 2003, um 14.00 Uhr im Sammelkasten am Lehrstuhl.

Lehrstuhl für Informatik I
Prof. Juraj Hromkovič
Dr. Hans-J. Böckenhauer
Dirk Bongartz

Musterlösung zur 2. Übung zur Vorlesung Algorithmische Grundlagen der Bioinformatik

Aachen, den 22. Mai 2003

Lösung zu Aufgabe 5

Sei $t = b^n$ und $p = a^m$.
Bei dem Vergleich von $t_i \dots t_{i+m-1}$ mit p ergibt sich stets ein passender Suffix der Länge 0, also ergibt die Good-Suffix-Regel nach einem durchgeführten Vergleich eine Verschiebung um eine Position. Insgesamt werden also bei alleiniger Anwendung der Good-Suffix-Regel $O(n-m) = O(n)$ Vergleiche durchgeführt.
Die Bad-Character-Regel ergibt aber in jedem Fall eine Verschiebung um m Positionen, da das Symbol b in p nicht vorkommt, insgesamt ergeben sich hier also $O(\frac{n}{m})$ Vergleiche.

Lösung zu Aufgabe 6

Sei $\mathcal{S}$ eine Multimenge mit m Symbolen über einem Alphabet Σ. Ein effizienter Algorithmus, der in einem String t der Länge n über Σ alle Teilstrings der Länge $m = |\mathcal{S}|$ findet, die aus genau den Symbolen in $\mathcal{S}$ bestehen, kann wie folgt arbeiten. Dabei setzen wir im Folgenden voraus, dass $m \leq n$ gilt (sonst kann es keinen Teilstring der Länge m in t geben).
Die Idee des Algorithmus besteht darin, von links nach rechts ein Fenster der Länge m über den Text t zu schieben und jeweils die Anzahlen der Vorkommen der Symbole aus Σ in diesem Fenster zu zählen und mit den entsprechenden Anzahlen in der Multimenge $\mathcal{S}$ zu vergleichen. Dabei lassen sich die Zähler beim Verschieben des Fensters von der Position i zur Position $i+1$ einfach dadurch aktualisieren, dass man den Zähler für das Symbol $a = t_i$ um 1 erniedrigt und den Zähler für $b = t_i + m$ um 1 erhöht.

Eingabe: Ein String $t = t_1 \dots t_n$ über dem Alphabet Σ und eine Multimenge $\mathcal{S} = \{a_1, \dots, a_m\}$ von Symbolen aus Σ.

1. Zähle die Anzahl der Vorkommen aller Symbole aus Σ in $\mathcal{S}$:
   for all $a \in \Sigma$ **do**
   $Anzahl(a) := 0$
   for $i := 1$ **to** m **do**
   $Anzahl(a_i) := Anzahl(a_i) + 1$

2. Initialisiere die Menge der Positionen, an denen in t einer der gesuchten Teilstrings beginnt, sowie die Zähler für die Symbolhäufigkeiten:
   $I := \emptyset$
   for all $a \in \Sigma$ **do**
   $Zaehl(a) := 0$

3. Bestimme die Anzahlen der Vorkommen für alle Symbole in Σ für jeden Teilstring der Länge m von t:
   for $i := 1$ **to** $n - m + 1$ **do** {Alle möglichen Anfangspositionen von Teilstrings der Länge m}
   if $i = 1$ **then** {Erster möglicher Teilstring, zähle die Vorkommen der Symbole}
   for $j := 1$ **to** m **do**
   $Zaehl(t_j) := Zaehl(t_j) + 1$
   else {Aktualisiere die Zähler}
   $Zaehl(t_{i-1}) := Zaehl(t_{i-1}) - 1$
   $Zaehl(t_{i+m-1}) := Zaehl(t_{i+m-1}) + 1$
   $test :=$ **true** {Vergleiche den Zähler mit der Multimenge $\mathcal{S}$}
   for all $a \in \Sigma$ **do**
   if $Anzahl(a) \neq Zaehl(a)$ **then**
   $test :=$ **false**
   if $test =$ **true then**
   $I := I \cup \{i\}$

Ausgabe: Die Menge I aller Positionen in t, an denen ein Teilstring beginnt, der genau aus den Symbolen in $\mathcal{S}$ besteht.

Es ist offensichtlich, dass der Algorithmus genau das Geforderte leistet. Da der Algorithmus nach der Initialisierung von *Anzahl* den Text nur einmal durchläuft, und pro gelesenem Zeichen maximal $|\Sigma|$ Vergleiche benötigt, liegt seine Laufzeit in $O(m + n \cdot |\Sigma|) = O(n \cdot |\Sigma|)$.
Für diese Laufzeitabschätzung sind wir davon ausgegangen, dass jeder Vergleich der beiden Zähler $Anzahl(a)$ und $Zaehl(a)$ in konstanter Zeit möglich ist. Falls diese Annahme unrealistisch wird, da die gegebene Multimenge sehr groß wird, ergibt sich hier noch ein zusätzlicher Faktor von $O(\log m)$.

Lösung zu Aufgabe 7

Ein Algorithmus, der mit Hilfe eines Suffix-Baums effizient die Ziv-Lempel-komprimierte Darstellung $ZL(t)$ eines Strings t berechnet, kann wie folgt arbeiten:
Der Algorithmus beruht auf der folgenden Idee: Wenn man schon einen Teil des Textes (bis zur Position $i-1$) komprimiert hat, sucht man in dem Suffix-Baum einen Pfad von der Wurzel aus mit der Beschriftung $Prior(i)$, also einen möglichst langen Präfix von $t_i \dots t_n$, der als Teilstring in $t_1 \dots t_{i-1}$ vorkommt. Der Wert s_i entspricht dann der kleinsten Blattbeschriftung in dem Teilbaum unterhalb dieses Pfades, und der Wert l_i entspricht der Länge der Pfadbeschriftung.

Eingabe: Ein Text $t = t_1 \dots t_n$.

1. Konstruiere einen kompakten Suffix-Baum T_t für t mit der Wurzel r.

2. Beschrifte jeden inneren Knoten v von T_t mit der kleinsten Blattbeschriftung in dem Teilbaum mit der Wurzel v, sei $c(v)$ diese Beschriftung.

3. Initialisiere die Ziv-Lempel-Komprimierung des Textes:
   $ZL(t) := \lambda$

4. Komprimiere den Text:

   $i := 1$ {Startposition des noch zu komprimierenden Teils des Textes}
   $laenge := 0$ {Länge des bisher komprimierten Teils des Textes}
   repeat
     $v := r$
     while (es gibt ein Kind x von v, so dass $pathlabel(x)$ ein Präfix von $t_i \dots t_n$ ist) **and** $(depth(x) \leq laenge - c(x))$ **do**
       $v := x$
     $s(i) := c(v)$
     $l(i) := depth(v)$
     if es gibt ein Kind x von v, so dass $label(v, x)$ und $t_{i+depth(v)} \dots t_n$ ein nichtleeres gemeinsames Präfix haben **then** {maximales Präfix $Prior(i)$ endet mitten in einer Kantenbeschriftung}
       $w :=$ maximales gemeinsames Präfix von $label(v, x)$ und $t_{i+depth(v)} \dots t_n$, so dass $(depth(x) + |w| \leq laenge - c(x))$
       $s(i) := c(x)$
       $l(i) := l(i) + |w|$
     if $l(i) > 0$ **then** $\{Prior(i) \neq \lambda\}$
       $ZL(t) := ZL(t) \cdot (s(i), l(i))$
       $laenge := laenge + l(i)$
       $i := i + l(i)$
     else $\{Prior(i) = \lambda\}$
       $ZL(t) := ZL(t) \cdot t_i$
       $laenge := laenge + 1$
       $i := i + 1$
   until $i > n$

Ausgabe: Der Ziv-Lempel-komprimierte Text $ZL(t)$.

Die Konstruktion des kompakten Suffix-Baums und die Berechnung der Werte $c(v)$ ist in einer Zeit in $O(n \log n)$ möglich, die eigentliche Komprimierung läuft dann in linearer Zeit. Also ergibt sich insgesamt eine Laufzeit in $O(n \log n + n) = O(n \log n)$.

Lösung zu Aufgabe 8

(a) Um aus einem einfachen Suffix-Baum für t das Suffix-Array Pos für t zu konstruieren, führt man auf dem Suffix-Baum eine Tiefensuche durch, bei der jeweils die Beschriftung der besuchten Blätter ausgegeben wird. Dabei besucht man die Kinder eines Knotens in der durch die lineare Ordnung $\prec$ gegebenen Reihenfolge der Kantenbeschriftungen, hierbei gelte $\$ \prec a$ für alle $a \in \Sigma$.

Die Reihenfolge, in der die Blattbeschriftungen ausgegeben werden, gibt die Reihenfolge an, in der die Zahlen im Array Pos gespeichert werden müssen.

(b) Um effizient zu entscheiden, ob ein Muster p Teilstring eines Strings t ist, kann man einfach eine binäre Suche über dem Suffix-Array Pos durchführen. Da bei der binären Suche auf einer Menge mit n sortierten Elementen maximal $\lceil \log n \rceil$ Schritte benötigt werden, um ein Element zu finden, kann man die Suche nach spätestens dieser Anzahl von Schritten abbrechen. In jedem dieser Schritte muss das Muster mit einem entsprechenden Teilstring der Länge m des Textes verglichen werden, insgesamt ergibt sich also eine Laufzeit in $O(m \cdot \log n)$.

Lehrstuhl für Informatik I
Prof. Juraj Hromkovič
Dr. Hans-J. Böckenhauer
Dirk Bongartz

3. Übung zur Vorlesung Algorithmische Grundlagen der Bioinformatik

Aachen, den 21. Mai 2003

Aufgabe 9

Seien $s = s_1 \dots s_m$ und $t = t_1 \dots t_n$ zwei Sequenzen, sei $1 < n \leq m$. Bestimmen Sie die Anzahl möglicher Alignments von s und t.
Hinweis: Bestimmen Sie zunächst für alle i mit $m \leq i \leq n + m$ die Anzahl der Alignments der Länge i. **10 Punkte**

Aufgabe 10

Seien die Strings u, v und t über einem Alphabet Σ gegeben. Wir sagen, dass t eine *Verflechtung* von u und v ist, wenn es Zerlegungen $u = u_1 \cdot \ldots \cdot u_k$ und $v = v_1 \cdot \ldots \cdot v_k$ gibt mit $u_1, \ldots, u_k, v_1, \ldots, v_k \in \Sigma^*$, so dass gilt

$$t = u_1 v_1 u_2 v_2 \dots u_k v_k.$$

(a) Geben Sie einen Algorithmus an, der für gegebene Strings u, v und t möglichst effizient entscheidet, ob t eine Verflechtung von u und v ist.

(b) Geben Sie einen Algorithmus an, der für gegebene Strings u, v und t möglichst effizient entscheidet, ob t eine Verflechtung von u und v als Teilstring enthält.

10 Punkte

Aufgabe 11

Wir betrachten eine Alignment-Bewertung δ mit $goal_\delta = \max$ und affiner Lückenbewertung, die eine Lücke der Länge k mit $-(\varrho + \sigma k)$ bewertet, wobei $\varrho, \sigma > 0$.

(a) Geben Sie eine geeignete Modifikation des Edit-Graphen an, so dass man das bezüglich δ optimale Alignment von zwei Strings s und t durch die Bestimmung eines kürzesten Pfades in dem modifizierten Edit-Graphen berechnen kann.

(b) Geben Sie eine Rekursionsformel an, mit deren Hilfe man mit dynamischer Programmierung das optimale Alignment bezüglich δ von zwei Strings $s = s_1 \dots s_m$ und $t = t_1 \dots t_n$ in einer Zeit in $O(m \cdot n)$ bestimmen kann.

10 Punkte

Aufgabe 12

Seien $s = s_1 \dots s_m$ und $t = t_1 \dots t_n$ zwei Strings über einem Alphabet Σ, sei $k \in \mathbb{N}$. Geben Sie einen Algorithmus an, der in einer Zeit in $O(k \cdot (n+m))$ entscheidet, ob ein globales Alignment von s und t existiert, das höchstens k Mismatches oder Lückensymbole enthält. **10 Punkte**

Abgabe: bis Mittwoch, den 28. Mai 2003, um 14.00 Uhr im Sammelkasten am Lehrstuhl.

Lehrstuhl für Informatik I
Prof. Juraj Hromkovič
Dr. Hans-J. Böckenhauer
Dirk Bongartz

Musterlösung zur 3. Übung zur Vorlesung Algorithmische Grundlagen der Bioinformatik

Aachen, den 2. Juni 2003

Lösung zu Aufgabe 9

Gegeben seien die zwei Sequenzen $s = s_1 \dots s_m$ und $t = t_1 \dots t_n$ mit $1 < n \leq m$. Betrachten wir zunächst die Anzahl möglicher Alignments der Länge $m+i$ mit $0 \leq i \leq n$. Sei (s', t') ein solches Alignment von s und t. Dann gibt es i Lückenpositionen in s' und somit $\binom{m+i}{i}$ Möglichkeiten, diese Lückenpositionen auszuwählen.
Für jede solche Auswahl von Lückenpositionen in s' bestimmen wir nun die Anzahl der Möglichkeiten, die Lücken in t' zu positionieren. In t' gibt es $m+i-n$ Lückensymbole, die wir beliebig auf diejenigen m Positionen verteilen können, an denen in s' keine Lücke steht. Hierfür gibt es also $\binom{m}{m+i-n}$ Möglichkeiten.
Die Anzahl aller möglichen Alignments der Länge $m+i$ ergibt sich somit als

$$\binom{m+i}{i}\binom{m}{m+i-n}.$$

Summation über alle möglichen Werte von i ergibt dann als Gesamtanzahl aller möglichen Alignments

$$\sum_{i=0}^{n} \binom{m+i}{i}\binom{m}{m+i-n}.$$

Lösung zu Aufgabe 10

(a) Seien die Strings $t = t_1 \dots t_l$, $u = u_1 \dots u_n$ und $v = v_1 \dots v_m$ gegeben. Es gelte ohne Beschränkung der Allgemeinheit $n \leq m$. Wir geben einen Algorithmus an, der mit Hilfe dynamischer Programmierung entscheidet, ob t eine Verflechtung von u und v ist. Dazu stellen wir eine $(l+1)\times(n+1)$-Matrix M auf, die an der Stelle (i,j) genau dann den Wert 1 enthält, wenn sich $t_1 \dots t_i$ als Verflechtung der beiden Strings $u_1 \dots u_j$ und $v_1 \dots v_{i-j}$ darstellen lässt.

Bei der Berechnung der Matrix M können wir auf die Berechnung aller Werte (i,j) mit $i < j$ verzichten, weil ein String t der Länge i sich offensichtlich nicht als Verflechtung eines Strings u der Länge $j > i$ und eines beliebigen weiteren Strings darstellen lässt.

Für die Matrix M ergibt sich die folgende Rekurrenz:

$$M(i,j) = \begin{cases} M(i-1,j-1), & \text{falls } t_i = u_j \text{ und } t_i \neq v_{i-j+1} \\ M(i-1,j), & \text{falls } t_i = v_{i-j} \text{ und } t_i \neq u_{j+1} \\ \max\{M(i-1,j-1), M(i-1,j)\} & \text{falls } t_i = u_j \text{ und } t_i = v_{i-j+1} \\ 0, & \text{sonst} \end{cases}$$

Initialisiert wird die Matrix M mit dem Wert

$$M(0,0) = 1,$$

da der leere String λ offenbar eine Verflechtung der beiden Strings λ und λ ist. Das Ergebnis lässt sich dann aus dem Matrixeintrag $M(|t|,|u|)$ ablesen, falls dieser Eintrag gleich 1 ist, dann existiert eine Verflechtung von u und v zu t, sonst nicht.

Man beachte, dass die oben angeführte Referenz es formal erfordert, auch auf einige der Matrixeinträge oberhalb der Diagonalen (also auf Felder (i, j) mit $i < j$) zurückzugreifen. Dieses Problem kann dadurch gelöst werden, dass man zu Beginn alle Einträge der Matrix mit dem Wert 0 initialisiert.

Aus den oben gemachten Ausführungen ergibt sich der folgende Algorithmus:

Eingabe: Strings u, v und t über dem Alphabet Σ
  if $|t| \neq |u| + |v|$ **then**
    return „NEIN“
  for $i := 0$ **to** $|t|$ **do**
    for $j = 0$ **to** $|u|$ **do**
      $M[i, j] := 0$
  $M[0, 0] = 1$ {Initialisiere die Matrix}
  for $i := 1$ **to** $|t|$ **do**
    for $j := 0$ **to** i **do**
      if $t_i = u_j$ **and** $t_i = v_{i-j+1}$ **then**
        $M[i, j] := \max\{M[i-1, j-1], M[i-1, j]\}$
      else if $t_i = u_j$ **and** $t_i \neq v_{i-j+1}$ **then**
        $M[i, j] := M[i-1, j-1]$
      else if $t_i = v_{i-j}$ **and** $t_i \neq u_{j+1}$ **then**
        $M[i, j] := M[i-1, j]$
Ausgabe: „JA“, falls $M(|t|, |u|) = 1$, „NEIN“ sonst.

Da sich jeder der Matrix-Einträge offenbar in konstanter Zeit berechnen lässt, benötigt der Algorithmus insgesamt eine Zeit in $O((|u| + |v|) \cdot |u|)$.

(b) Um für gegebene Strings u, v und t zu überprüfen, ob t eine Verflechtung von u und v als Teilstring enthält, führen wir den in Aufgabenteil (a) vorgestellten Algorithmus für jeden Teilstring der Länge $|u| + |v|$ von t durch. Falls sich ein Teilstring $t_i \dots t_{i+|u|+|v|}$ als Verflechtung von u und v darstellen lässt, wird dieses bei der i-ten Ausführung des Algorithmus gefunden. Es ergibt sich folglich eine Gesamtlaufzeit des Algorithmus' von $O((|t| - |u| - |v|) \cdot ((|u| + |v|) \cdot |u|)) = O(|t|^2 \cdot |u|)$.

Lösung zu Aufgabe 11

(a) Bei der *affinen Lückenbewertung* entspricht der Wert $\rho > 0$ der „Strafe“ für die Einfügung der Lücke und der Wert $\sigma > 0$ der „Strafe“ für jedes einzelne Symbol der Lücke. Zur Anpassung des aus der Vorlesung bekannten Edit-Graphen an die neue Lückenbewertung modifiziere zunächst die Gewichtsfunktion $c : E \to \mathbb{Q}$ des Edit-Graphen wie folgt.

$$\begin{aligned} c'((i,j),(i,j+1)) = c'((i,j),(i+1,j)) &= (\rho + \sigma) \\ c'((i,j),(i+1,j+1)) &= -p(s_{i+1}, t_{j+1}) \quad \text{für alle } i, j \end{aligned}$$

Erweitere schließlich den modifizierten Edit-Graphen durch Einfügung längerer vertikaler und horizontaler Kanten, d.h.

$$\begin{aligned} E' &:= E \\ &\cup \{((i,j),(i+k,j)) \mid \forall i, j, 1 < k \leq m - i\} \\ &\cup \{((i,j),(i,j+k)) \mid \forall i, j, 1 < k \leq n - i\} \end{aligned}$$

Versehe eine beliebige neu eingefügte Kante $e = ((i,j),(i+k,j))$ bzw. $e' = ((i,j),(i,j+k))$ mit dem Gewicht $c'(e) = c'(e') = (\rho + k\sigma)$.

Da wir im Wesentlichen die aus der Vorlesung bekannte Bewertungsfunktion (mit umgekehrtem Vorzeichen) verwendet haben, und darüber hinaus die affine Lückenbewertung angemessen berücksichtigt haben, können wir ein optimales Alignment nun durch Bestimmung eines kürzesten Pfades von dem Knoten $(0, 0)$ zum Knoten (m, n) ermitteln.

(b) Betrachte die folgenden drei Rekurrenzen:

$$
\begin{aligned}
M^{\downarrow}(i,j) &= \max \begin{cases} M^{\downarrow}(i-1,j) - \sigma \\ M(i-1,j) - (\rho + \sigma) \end{cases} \\
M^{\rightarrow}(i,j) &= \max \begin{cases} M^{\rightarrow}(i,j-1) - \sigma \\ M(i,j-1) - (\rho + \sigma) \end{cases} \\
M(i,j) &= \max \begin{cases} M(i-1,j-1) + p(s_i, t_j) \\ M^{\downarrow}(i,j) \\ M^{\rightarrow}(i,j) \end{cases}
\end{aligned}
$$

Der Wert $M^{\downarrow}(i,j)$ entspricht dann der Bewertung eines Alignments der beiden Strings $s_1 \dots s_i$ und $t_1 \dots t_j$, welches mit einer Deletion, d.h. einer Lücke in t endet, während der Wert $M^{\rightarrow}(i,j)$ die Bewertung eines Alignments berechnet, welches mit einer Insertion endet, d.h. einer Lücke in s. Der erste Ausdruck in den Rekurrenzen $M^{\downarrow}(i,j)$ und $M^{\rightarrow}(i,j)$ entspricht dann einer Erweiterung der Lücke, während der zweite Ausdruck dem Beginn einer Lücke entspricht. Die gleichzeitige Berechnung der drei Matrizen M, $M^{\downarrow}$ und $M^{\rightarrow}$ ist in einer Zeit in $O(m \cdot n)$ möglich, da jeder einzelne Eintrag in jeder dieser Matrizen in konstanter Zeit berechenbar ist.

Lösung zu Aufgabe 12

Betrachte als Bewertungsfunktion für das gesuchte Alignment die *Edit-Distanz*. Man beachte, dass diese Funktion gerade die Anzahl von Mismatches und Lücken zählt, da sie diesen jeweils den Wert 1 zuordnet, während Matches nicht gezählt, d.h. mit 0 bewertet werden.
Wollen wir nun wissen, ob ein globales Alignment zweier Strings $s = s_1 \dots s_m$ und $t = t_1 \dots t_n$ existiert, welches höchstens k Mismatches oder Lückensymbole enthält, betrachten wir den Wert in Feld $M(m,n)$ der Alignment-Matrix M und antworten „Ja“, falls dieser Wert kleiner oder gleich k ist, und „Nein“, falls der eingetragene Wert größer als k ist.
Wie wir jedoch aus der Vorlesung wissen, benötigt die Berechnung der vollständigen Alignment-Matrix unter der sinnvollen Annahme, dass $k < \max\{m,n\}$ gilt, mehr Zeit als $O(k \cdot (m+n))$. Betrachten wir nun ein beliebiges Feld (i,j) der Alignment-Matrix mit $i + k < j$, d.h. das Feld ist in der Horizontalen mehr als k Felder von der Diagonale (j,j) der Matrix entfernt. Dann enthält jedes zugehörige Alignment, das die Positionen s_i und t_j einander zuordnet, mehr als k Lückensymbole und trägt somit nicht zur Beantwortung der Frage, ob ein Alignment mit weniger als k Lückensymbolen existiert, bei. Insbesondere folgt aus dieser Argumentation auch, dass für jede sinnvolle Eingabe $|m-n| \leq k$ gelten muss.
Aus diesem Grund können bei der Berechnung der Alignment-Matrix alle Werte, die mehr als k Felder von der Diagonale der Matrix entfernt sind, ignoriert werden, und wir berechnen nur einen $2k$ Felder breiten „Streifen“ um die Diagonale herum. Die Werte (i,j) der Felder, die genau an der Grenze dieses „Streifens“ liegen, und zu deren Berechnung somit ein Wert fehlt (entweder der Wert $(i-1,j)$, falls (i,j) an der rechten Grenze des Streifens liegt, oder der Wert $(j-1,i)$, falls (i,j) an der linken Grenze liegt), errechnen wir aus dem Minimum der beiden vorhandenen Werte.
Durch diese Beschränkung der Alignment-Matrix können wir die Laufzeit der Berechnung der Matrix auf $O(2k \cdot \max\{m,n\}) = O(k \cdot (m+n))$ drücken.

Lehrstuhl für Informatik I
Prof. Juraj Hromkovič
Dr. Hans-J. Böckenhauer
Dirk Bongartz

4. Übung zur Vorlesung Algorithmische Grundlagen der Bioinformatik

Aachen, den 28. Mai 2003

Aufgabe 13

Seien zwei Strings $s = s_1 \dots s_m$ und $t = t_1 \dots t_n$ gegeben. Wir nennen ein (globales) Alignment A von s und t ε-*optimal*, falls die Bewertung von A nur um eine additive Konstante ε von der optimalen Alignment-Bewertung abweicht.
Geben Sie einen möglichst effizienten Algorithmus an, der für jedes Paar (i, j) mit $1 \leq i \leq m$ und $1 \leq j \leq n$ ausgibt, ob es ein ε-optimales Alignment von s und t gibt, so dass s_i und t_j in derselben Spalte des Alignments stehen.
Hinweis: Bestimmen Sie zunächst die Alignment-Matrizen für s und t sowie für $s^R = s_m \dots s_1$ und $t^R = t_n \dots t_1$. **10 Punkte**

Aufgabe 14

Es sei ein multiples Alignment A der Länge l für eine Menge S von Strings über einem Alphabet Σ gegeben. Ein *Profil* von A ist eine Abbildung, die jeder Spalte von A und jedem Symbol in $\Sigma \cup \{-\}$ die relative Häufigkeit zuordnet, mit der dieses Symbol in dieser Spalte vorkommt.
Zum Beispiel ergibt sich für das Alignment

```
A C G – T
A C A C T
A G G C –
G C – C C
```

das in der folgenden Tabelle dargestellte Profil:

	C_1	C_2	C_3	C_4	C_5
A	0.75	0	0.25	0	0
C	0	0.75	0	0.75	0
G	0.25	0.25	0.5	0	0.25
T	0	0	0	0	0.5
–	0	0	0.25	0.25	0.25

Dabei bezeichnen $C_1, \dots, C_5$ die Spalten des Alignments.
Ein Profil eines Alignments kann man auffassen als eine kompakte Beschreibung einer Familie von Sequenzen. Wir wollen nun eine weitere Sequenz mit einem gegebenen Profil vergleichen, um festzustellen, ob auch diese Sequenz zu der Familie gehört oder nicht.
Entwerfen Sie einen effizienten Algorithmus, der einen gegebenen String mit dem Profil eines multiplen Alignments vergleicht und eine Bewertung ausgibt, die den Grad der Übereinstimmung beschreibt. **10 Punkte**

(bitte wenden)

Aufgabe 15

Das *Longest-Common-Subsequence-Problem* ist das folgende Optimierungsproblem:

Eingabe: Eine Menge von k Strings $s_1, \ldots, s_k$ über einem Alphabet Σ.

Zulässige Lösungen: Jeder String $t \in \Sigma^*$, der eine gemeinsame Teilsequenz von allen $s_1, \ldots, s_k$ ist.

Kosten: Für eine zulässige Lösung t gilt $cost(t) = |t|$.

Optimierungsziel: Maximierung.

(a) Zeigen Sie, wie man das Longest-Common-Subsequence-Problem auf die Bestimmung eines optimalen multiplen Alignments zurückführen kann.

(b) Zeigen Sie, wie man das Shortest-Common-Supersequence-Problem auf die Bestimmung eines optimalen multiplen Alignments zurückführen kann.

10 Punkte

Abgabe: bis Mittwoch, den 4. Juni 2003, um 14.00 Uhr im Sammelkasten am Lehrstuhl.

Lehrstuhl für Informatik I
Prof. Juraj Hromkovič
Dr. Hans-J. Böckenhauer
Dirk Bongartz

Musterlösung zur 4. Übung zur Vorlesung Algorithmische Grundlagen der Bioinformatik

Aachen, den 6. Juni 2003

Lösung zu Aufgabe 13

Seien zwei Strings $s = s_1 \dots s_m$ und $t = t_1 \dots t_n$ gegeben. Der folgende Algorithmus entscheidet für jedes Paar (i, j) mit $1 \leq i \leq m$ und $1 \leq j \leq n$, ob ein ε-optimales Alignment von s und t existiert, bei dem s_i und t_j in derselben Spalte stehen. Der Algorithmus beruht auf der folgenden Idee: Wenn für ein Paar (i, j) ein solches Alignment existiert, dann lässt sich dieses Alignment zerlegen in ein Alignment von $s_1 \dots s_{i-1}$ und $t_1 \dots t_{j-1}$, die Spalte mit s_i und t_j und ein Alignment von $s_{i+1} \dots s_m$ und $t_{j+1} \dots t_n$. Um die Kosten aller dieser Alignments von Präfixen bzw. Suffixen von s und t zu bestimmen, reicht es aus, zwei Alignment-Matrizen zu berechnen.

Eingabe: Zwei Strings $s = s_1 \dots s_m$ und $t = t_1 \dots t_n$.

1. Bestimme die Alignment-Matrizen M_1 für s und t sowie M_2 für $s^R = s_m \dots s_1 = s'_1 \dots s'_m$ und $t^R = t_n \dots t_1 = t'_1 \dots t'_n$. Bestimme daraus die optimale Alignmentbewertung c für s und t. (Dabei kann man das Optimum in beiden Matrizen ablesen, da die optimale Alignmentbewertung für s und t die gleiche ist wie für s^R und t^R.) Initialisiere eine Ergebnis-Matrix *Erg*, in der an der Stelle (i, j) eingetragen wird, ob es ein ε-optimales Alignment von s und t gibt, bei dem s_i und t_j in derselben Spalte stehen.

2. Für alle (i, j), $1 \leq i \leq m, 1 \leq j \leq n$ gehe dann wie folgt vor:

   2.1 Bestimme die Bewertung c' für das optimale Alignment von s und t, bei dem s_i und t_j in derselben Spalte stehen, als $c' = M_1(i-1, j-1) + p(s_i, t_j) + M_2(m-i, n-j)$.

   2.2 Falls $c + \varepsilon \leq c'$, trage in $Erg(i, j)$ „JA“ ein, sonst trage „NEIN“ ein.

Ausgabe: Die Matrix *Erg*.

Wir analysieren nun noch die Laufzeit des Algorithmus: Die Erstellung der Alignment-Matrizen in Schritt 1 benötigt eine Zeit in $O(2(n \cdot m)) \in O(n \cdot m)$. Die Laufzeit von Schritt 2 liegt ebenfalls in $O(n \cdot m)$, da für jedes Paar (i, j), $1 \leq i \leq m, 1 \leq j \leq n$ nur eine konstante Anzahl von Schritten durchgeführt wird. Insgesamt ergibt sich somit eine Laufzeit in $O(n \cdot m)$.

Lösung zu Aufgabe 14

Um einen gegebenen String $v = v_1 \dots v_n$ mit einem gegebenen Profil P zu vergleichen, erweitern wir das in der Vorlesung vorgestellte Alignment-Verfahren und bestimmen ein Alignment von v mit dem Profil P.
Seien $C_1 \dots C_m$ die Spalten des Profils, sei $s(a, j)$ die relative Häufigkeit von $a \in \Sigma$ in der Spalte j für $1 \leq j \leq m$. Desweiteren sei eine Alignment-Bewertung δ mit Optimierungsziel Maximierung gegeben. Dann seien

$$S(a, j) = \begin{cases} \sum_{b \in \Sigma} p(a, b) \cdot s(b, j), & \text{falls } a \neq -, \\ \sum_{b \in \Sigma} g \cdot s(b, j), & \text{sonst.} \end{cases}$$

die Kosten für das Alignment des Symbols a mit der Spalte j des Profils.
Wir berechnen jetzt eine $((n+1) \times (m+1))$-Alignment-Matrix M wie folgt:

- Initialisierung der ersten Zeile: $M(0,0) = 0$ und $M(0,j) = M(0,j-1) + S(-,j)$ für $1 \leq j \leq m$.
- Initialisierung der ersten Spalte: $M(i,0) = i \cdot g$.
- Für alle $i,j > 0$ verwenden wir die folgende Rekurrenz:

$$M(i,j) = \max\{M(i-1,j-1) + S(v_i,j), M(i-1,j) + g, M(i,j-1) + S(-,j)\}.$$

$M(n,m)$ ist dann die Bewertung des optimalen Alignments von v mit dem Profil und beschreibt den Grad der Übereinstimmung der beiden. Die Bestimmung eines Wertes $S(a,j)$ ist offenbar in einer Zeit in $O(|\Sigma|)$ möglich, damit benötigt der gesamte Algorithmus eine Zeit in $O(|\Sigma| \cdot n \cdot m)$.

Lösung zu Aufgabe 15

(a) Sei ein multiples Alignment für eine Menge S von Strings gegeben. Die Folge aller Symbole in denjenigen Spalten dieses multiplen Alignments, die weder Mismatches noch Lücken enthalten, beschreibt eine gemeinsame Teilsequenz der Strings aus S. Damit lässt sich eine optimale Lösung für das Longest-Common-Subsequence-Problem wie folgt berechnen:

Eingabe: Eine Menge von k Strings $S = \{s_1, \ldots, s_k\}$ über dem Alphabet Σ.

1. Definiere eine Spaltenbewertung δ mit Optimierungsziel Maximierung für das multiple Alignment von k Strings, die jede Spalte mit 1 bewertet, wenn sie aus genau k gleichen Symbolen aus Σ besteht, und mit 0 sonst.
2. Berechne ein optimales multiples Alignment A für S bezüglich δ.
3. Sei m die Anzahl der Spalten von A. Für alle $1 \leq j \leq m$ setze $c_j := a$, falls die Spalte j von A genau k-mal das Symbol $a \in \Sigma$ enthält, und $c_j := \lambda$ sonst.
4. Setze $c := c_1 \ldots c_m$.

Ausgabe: Die längste gemeinsame Teilsequenz c von $s_1, \ldots, s_k$.

Da ein optimales Alignment bezüglich der Bewertung δ eine maximale Anzahl von Spalten ohne Mismatches oder Lücken enthält, bestimmt dieser Algorithmus also eine optimale Lösung für das Longest-Common-Subsequence-Problem.

(b) Sei ein multiples Alignment für eine Menge S von Strings über einem Alphabet Σ gegeben. Aus diesem Alignment lässt sich eine Supersequenz für S wie folgt ablesen: Für jede Spalte des Alignments, in der genau die verschiedenen Symbole $a_1, \ldots, a_i \in \Sigma$ vorkommen, betrachte den String $a_1 \ldots a_i$. Die Konkatenation dieser Strings für alle Spalten ergibt dann eine Supersequenz für S. Also lässt sich eine kürzeste Supersequenz durch ein Alignment berechnen, bei dem die Bewertungsfunktion so definiert ist, dass die Anzahl der verschiedenen Symbole pro Spalte möglichst klein wird. Damit ergibt sich der folgende Algorithmus für das Shortest-Common-Supersequence-Problem:

Eingabe: Eine Menge von k Strings $S = \{s_1, \ldots, s_k\}$ über dem Alphabet Σ.

1. Definiere eine Spaltenbewertung δ mit Optimierungsziel Minimierung, die jede Spalte mit der Anzahl der darin vorkommenden verschiedenen Symbole aus Σ bewertet.
2. Berechne ein optimales multiples Alignment A für S bezüglich δ.
3. Sei m die Anzahl der Spalten von A. Für alle $1 \leq j \leq m$ setze $c_j = a_1 \ldots a_i$, falls die Spalte j von A genau die Symbole $a_1, \ldots, a_i \in \Sigma$ (sowie gegebenenfalls noch Lückensymbole) enthält. Die Reihenfolge der Symbole $a_1, \ldots, a_i$ ist dabei beliebig.
4. Setze $c := c_1 \ldots c_m$.

Ausgabe: Die kürzeste gemeinsame Supersequenz c von $s_1, \ldots, s_k$.

Lehrstuhl für Informatik I
Prof. Juraj Hromkovič
Dr. Hans-J. Böckenhauer
Dirk Bongartz

5. Übung zur Vorlesung Algorithmische Grundlagen der Bioinformatik

Aachen, den 4. Juni 2003

Aufgabe 16

Sei eine Instanz $I = (A, B, C)$ des DDP mit $|A| = n$ und $|B| = m$ gegeben. Geben Sie eine (möglichst allgemeine) Bedingung an I an, so dass die folgende Aussage gilt:

> Wenn das DDP mit Eingabe I eine zulässige Lösung hat, dann gibt es mindestens $\left(\lfloor \frac{n}{m} \rfloor\right)!$ verschiedene zulässige Lösungen.

Begründen Sie Ihre Antwort. **10 Punkte**

Aufgabe 17

Zeigen Sie, dass das Disjoint-DDP, also die Variante des DDP, bei der die Restriktionsstellen der beiden verwendeten Restriktionsenzyme disjunkt sind, NP-schwer ist. Verwenden Sie hierfür eine Polynomzeit-Reduktion von dem 3-Partition-Problem auf die Entscheidungsvariante Dec-Disjoint-DDP des Disjoint-DDP. Dabei sei das 3-Partition-Problem wie folgt definiert:

Eingabe: Eine natürliche Zahl h und eine Menge $P = \{p_1, \ldots, p_{3n}\}$ natürlicher Zahlen mit $\sum_{i=1}^{3n} p_i = h \cdot n$ und $\frac{h}{4} < p_i < \frac{h}{2}$ für $1 \leq i \leq 3n$.

Ausgabe: „JA“, falls eine Partition von P in n dreielementige Teilmengen $\{p_{1,1}, p_{1,2}, p_{1,3}\}, \ldots,$ $\{p_{n,1}, p_{n,2}, p_{n,3}\}$ existiert, so dass $\sum_{j=1}^{3} p_{i,j} = h$ gilt für alle $1 \leq i \leq n$. „NEIN“ sonst.

Hinweis: Konstruieren Sie aus einer gegebenen Instanz P des 3-Partition-Problems eine Instanz des Dec-Disjoint-DDP, wobei Sie die Menge A derart wählen, dass A alle Werte aus P enthält und zusätzlich noch $n-1$ (relativ große) Werte. Für jede Lösung des Dec-Disjoint-DDP sollen dann die Werte in A gemäß einer Partition von P angeordnet sein, mit den zusätzlichen Werten als Trennung zwischen den einzelnen Mengen der Partition. **10 Punkte**

Aufgabe 18

Sei A eine Multimenge mit $\binom{k}{2}$ Elementen aus $\mathbb{N} - \{0\}$ und sei P eine zulässige Lösung des PDP mit Eingabe A. Zeigen Sie, dass für alle $1 \leq i \leq k-1$ gilt

$$\sum_{\alpha \in \text{level}_P(i)} \alpha = \sum_{\beta \in \text{level}_P(k-i)} \beta.$$

10 Punkte

Aufgabe 19

Sei A eine Multimenge mit $\binom{k}{2}$ Elementen aus $\mathbb{N} - \{0\}$ und seien P und P' zwei zulässige Lösungen des PDP mit Eingabe A. Beweisen oder widerlegen Sie die folgende Behauptung:

$$\text{level}_P(i) = \text{level}_{P'}(i) \quad \text{für alle } 1 \leq i \leq k-1.$$

10 Punkte

(bitte wenden)

Aufgabe 20

Wir betrachten die folgende Variante des Partial-Digest-Problems, bei der wir als Eingabe nicht nur die Multimenge aller Distanzen bekommen, sondern zusätzlich noch für jede Distanz die Information, zu welchem Level der Lösung sie gehören soll. Dieses Problem wollen wir *Level-PDP* nennen, es lässt sich formal wie folgt darstellen:

Eingabe: Eine Folge von $k-1$ Multimengen $L_1, \ldots, L_{k-1}$ von Elementen aus $\mathbb{N} - \{0\}$, so dass $|L_i| = k - i$ gilt für alle $1 \leq i \leq k-1$.

Ausgabe: Eine zulässige Lösung für das PDP mit der Distanzmenge $A = \bigcup_{i=1}^{k-1} L_i$, für die gilt, dass $L_i = \text{level}_P(i)$ für alle $1 \leq i \leq k-1$.

Geben Sie einen möglichst effizienten Algorithmus an, der das Level-PDP löst. **10 Punkte**

Abgabe: bis Mittwoch, den 18. Juni 2003, um 14.00 Uhr im Sammelkasten am Lehrstuhl.

Lehrstuhl für Informatik I
Prof. Juraj Hromkovič
Dr. Hans-J. Böckenhauer
Dirk Bongartz

Musterlösung zur 5. Übung zur Vorlesung Algorithmische Grundlagen der Bioinformatik

Aachen, den 23. Juni 2003

Lösung zu Aufgabe 16

Sei eine Instanz $I = (A, B, C)$ des DDP mit $|A| = n$ und $|B| = m$ gegeben. Es gelte o.B.d.A. $n \geq m$. Wenn nun als Bedingung

$$A = C$$

gilt, dann muss für die Anordnungen π von A und ϕ von B gelten, dass $Pos(\phi) \subseteq Pos(\pi)$, d. h. aus biologischer Sicht entsprechen die Restriktionsstellen des Enzyms $\mathcal{B}$ einigen von $\mathcal{A}$. Dann existiert zwischen zwei solchen Restriktionsstellen von B mindestens ein Abschnitt in A, der wenigstens $\lfloor \frac{n}{m} \rfloor$ Distanzen enthält, die nun beliebig permutiert werden können, da keine Distanz in B die beliebige Anordnung der Distanzen in diesem Abschnitt von A verhindert. Wenn das DDP bezüglich der Eingabe I jetzt eine zulässige Lösung besitzt, dann existieren zusätzlich zu dieser Lösung noch mindestens $(\lfloor \frac{n}{m} \rfloor)! - 1$ weitere Lösungen, die wie oben beschrieben durch Permutation der Distanzen in dem besagten Abschnitt in A entstehen.

Lösung zu Aufgabe 17

Wir wollen zeigen, dass das DISJOINT-DDP NP-schwer ist. Hierfür verwenden wir nicht die in dem Lösungshinweis angeregte Reduktion von dem 3-Partition-Problem, sondern eine etwas einfachere Reduktion vom SET-PARTITION-Problem (siehe dazu auch Definition 7.6 der Vorlesung) auf das zugehörige Entscheidungsproblem DEC-DISJOINT-DDP.
Sei dazu eine Instanz $X = \{x_1, \ldots, x_n\}$, $x_i \in \mathbb{N} - \{0\}$ für $1 \leq i \leq n$, des SET-PARTITION-Problems gegeben. Dann konstruieren wir daraus eine Instanz für das DEC-DISJOINT-DDP wie folgt:

- $A = \{x'_1, \ldots, x'_n\}$ mit $x'_i = 10 \cdot x_i$ für $2 \leq i \leq n$ and $x'_1 = 10 \cdot x_1 + 2$.
- $B = \{\frac{\alpha}{2}, \frac{\alpha}{2}\}$ mit $\alpha = 10 \cdot \sum_{x \in X} x + 2$.
- $C = \{1, x'_1 - 1, x'_2, \ldots, x'_n\}$

Wenn nun eine Lösung für das SET-PARTITION-Problem bezüglich der Eingabe X existiert, d. h. wenn es eine Zerlegung von X in zwei disjunkte Mengen Y und Z mit $\sum_{y \in Y} y = \sum_{z \in Z} z$ gibt, dann erhalten wir wie folgt eine Lösung für das DEC-DISJOINT-DDP mit Eingabe A, B und C. Wir nehmen dazu o.B.d.A. an, dass $x_1 = y_1 \in Y$ gilt. Dann bildet die Anordnung $\pi = (y'_2, y'_3, \ldots, y'_{|Y|}, y'_1, z'_1, \ldots, z'_{|Z|})$ von Distanzen aus A zusammen mit der einzig möglichen Anordnung ϕ von Distanzen aus B eine zulässige Lösung für das DEC-DISJOINT-DDP (siehe dazu auch Abbildung 1).
Wenn auf der anderen Seite eine Lösung für das DEC-DISJOINT-DDP-Problem mit Eingabe A, B und C existiert, gelten die folgenden Eigenschaften. Bezeichne $Pos(\pi)$ $[Pos(\phi)]$ die zu einer zulässigen Anordnung π $[\phi]$ der Distanzen aus A $[B]$ zugehörige Punktmenge. Aufgrund unserer Wahl der Distanzen aus A besitzt jede Position in der Punktmenge $Pos(\pi)$ einen Wert p, so dass entweder $p \equiv 0 \pmod{10}$ oder $p \equiv 2 \pmod{10}$ gilt. Weil $\frac{\alpha}{2}$ die einzige Position (außer den beiden Endpunkten) in der Punktmenge $Pos(\phi)$ ist, und $\frac{\alpha}{2} \equiv 1 \pmod{10}$ gilt, liegt die in C vorkommende Distanz 1 unmittelbar neben der Grenze $\frac{\alpha}{2}$. Da die Distanz x'_1 in C nicht vorhanden ist, muss für eine zulässige Lösung ein Teil der Größe 1 von x'_1 auf der einen Seite von $\frac{\alpha}{2}$ liegen und der Rest von x'_1 auf der anderen Seite. Folglich erhalten wir dieselbe

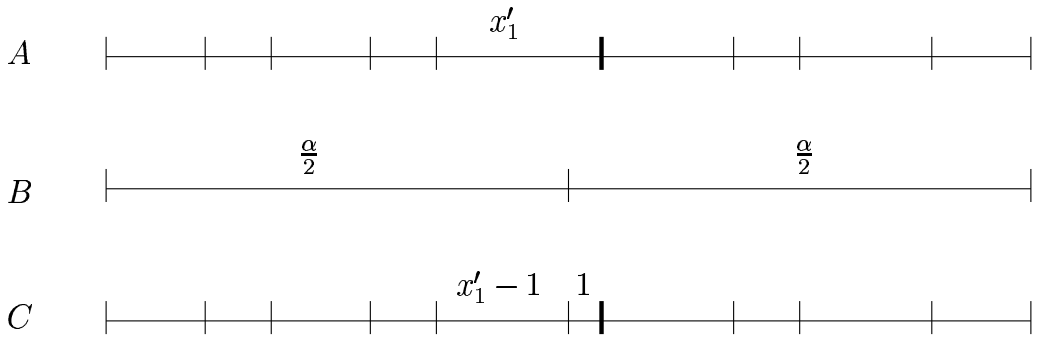


Abbildung 1: Reduktion von SET-PARTITION auf DEC-DISJOINT-DDP.

Situation wie in Abbildung 1. Daher können wir durch Partitionierung der Menge X bezüglich der Grenze $\frac{\alpha}{2}$ eine Lösung für das SET-PARTITION-Problem mit Eingabe X ableiten.
Hinweis: Eine Reduktion von dem 3-Partition-Problem, wie in dem Lösungshinweis zu dieser Aufgabe vorgeschlagen, ist ebenfalls möglich, hiermit kann man sogar zeigen, dass das Disjoint-DDP stark NP-schwer ist, dass also auch die eingeschränkte Variante des Problems NP-schwer bleibt, in der man nur Eingaben zulässt, in denen der Wert der größten Distanz polynomiell in der Länge der Eingabe ist. Diese Reduktion kann man in der folgenden Arbeit nachlesen:
M. Cieliebak, S. Eidenbenz, G. Woeginger: Double digest revisited: complexity and approximability in the presence of noisy data. Technischer Bericht 382, ETH Zürich, 2002.

Lösung zu Aufgabe 18

Sei A eine Multimenge mit $\binom{k}{2}$ Elementen aus $\mathbb{N} - \{0\}$ und sei P eine zulässige Lösung des PDP mit Eingabe A.
Wir zeigen, dass für alle $1 \leq i \leq k-1$ gilt: $\sum_{\alpha \in \text{level}_P(i)} \alpha = \sum_{\beta \in \text{level}_P(k-i)} \beta$.
Hierfür werden wir zeigen, dass jede atomare Distanz von gleich vielen Fragmenten aus $\text{level}_P(i)$ wie aus $\text{level}_P(k-i)$ überdeckt wird. Seien $d_1, \ldots, d_{k-1}$ die $k-1$ atomaren Distanzen in der Reihenfolge, in der sie in der Lösung P vorkommen.
Wir berechnen zunächst die Gesamtlänge aller Fragmente in $\text{level}_P(i)$. Es gibt genau $(k-1) - i + 1 = k - i$ Fragmente in $\text{level}_P(i)$, die jeweils genau i aufeinander folgende atomare Fragmente überdecken. Es ergibt sich also die Gesamtlänge

$$\sum_{\alpha \in \text{level}_P(i)} \alpha = \sum_{j=1}^{k-i} \sum_{l=0}^{i-1} d_{j+l} = \sum_{j=1}^{k-i} \sum_{l=1}^{i} d_{j+l-1}.$$

Analog lässt sich die Gesamtlänge aller Fragmente in $\text{level}_P(k-i)$ berechnen. Hierin gibt es genau $(k-1) - (k-i) + 1 = i$ Fragmente, die jeweils genau $k-i$ aufeinander folgende atomare Fragmente überdecken. Es ergibt sich also die Gesamtlänge

$$\sum_{\beta \in \text{level}_P(k-i)} \beta = \sum_{j=1}^{i} \sum_{l=0}^{(k-i)-1} d_{j+l} = \sum_{j=1}^{i} \sum_{l=1}^{k-i} d_{j+l-1}.$$

Offenbar gilt

$$\sum_{j=1}^{k-i} \sum_{l=1}^{i} d_{j+l-1} = \sum_{j=1}^{i} \sum_{l=1}^{k-i} d_{j+l-1},$$

woraus die Behauptung folgt.

Lösung zu Aufgabe 19

Sei A eine Multimenge mit $\binom{k}{2}$ Elementen aus $\mathbb{N} - \{0\}$ und seien P und P' zwei zulässige Lösungen des PDP mit Eingabe A. Die Behauptung

$$\text{level}_P(i) = \text{level}_{P'}(i)$$

ist falsch, wie das folgende Gegenbeispiel zeigt:
Sei $A = \{1, 1, 2, 2, 2, 3, 3, 4, 4, 5, 5, 5, 6, 7, 7, 7, 8, 9, 10, 11, 12\}$, dann sind $P = \{0, 1, 2, 5, 7, 9, 12\}$ und $P' = \{0, 1, 5, 7, 8, 10, 12\}$ zwei zulässige Lösungen für das PDP mit Eingabe A, wie die folgende Tabelle zeigt:

$P = \{0, 1, 2, 5, 7, 9, 12\}$	$P' = \{0, 1, 5, 7, 8, 10, 12\}$
$\text{level}_P(1) = \{1, 1, 3, 2, 2, 3\}$	$\text{level}_{P'}(1) = \{1, 4, 2, 1, 2, 2\}$
$\text{level}_P(2) = \{2, 4, 5, 4, 5\}$	$\text{level}_{P'}(2) = \{5, 6, 3, 3, 4\}$
$\text{level}_P(3) = \{5, 6, 7, 7\}$	$\text{level}_{P'}(3) = \{7, 7, 5, 5\}$
$\text{level}_P(4) = \{7, 8, 10\}$	$\text{level}_{P'}(4) = \{8, 9, 7\}$
$\text{level}_P(5) = \{9, 11\}$	$\text{level}_{P'}(5) = \{10, 11\}$
$\text{level}_P(6) = \{12\}$	$\text{level}_{P'}(6) = \{12\}$

Wie man aus der Tabelle erkennt, gilt zum Beispiel $4 \in \text{level}_{P'}(1)$, aber $4 \notin \text{level}_P(1)$, also folgt $\text{level}_P(1) \neq \text{level}_{P'}(1)$.

Lösung zu Aufgabe 20

Es ist bislang unbekannt, ob das Level-PDP in polynomieller Zeit lösbar ist oder nicht. Man kann aber die zusätzliche Information über die Levelzugehörigkeit der Distanzen dafür nutzen, den in der Vorlesung vorgestellten Backtracking-Algorithmus für das PDP (Algorithmus 7.1) zu beschleunigen. Dieser Backtracking-Algorithmus unter Ausnutzung der Level-Information benötigt offenbar höchstens soviel Zeit wie der Algorithmus 7.1 selbst. Während man aber für den Algorithmus 7.1 ein Worst-Case-Beispiel kennt, für das er eine exponentielle Zeit benötigt, kennt man bei zusäzlicher Level-Information kein solches Worst-Case-Beispiel.

Lehrstuhl für Informatik I
Prof. Juraj Hromkovič
Dr. Hans-J. Böckenhauer
Dirk Bongartz

6. Übung zur Vorlesung Algorithmische Grundlagen der Bioinformatik

Aachen, den 18. Juni 2003

Aufgabe 21

In der Vorlesung haben wir gesehen, wie man die Kartierung durch Hybridisierung mit eindeutigen Probes durchführen kann, indem man bestimmt, ob die Hybridisierungsmatrix die Consecutive-Ones-Eigenschaft (C1P) besitzt. Wir wollen nun zeigen, dass man dieses Problem auch graphtheoretisch formulieren kann mit Hilfe der so genannten Intervall-Graphen.
Ein *Intervall-Graph* ist definiert als der Schnittgraph einer Menge von reellen Intervallen, d. h. die Knoten entsprechen den Intervallen, und zwei Knoten sind genau dann durch eine Kante verbunden, wenn die zugehörigen Intervalle einen nichtleeren Schnitt besitzen.

(a) Ein Graph heißt *trianguliert*, wenn jeder einfache Kreis der Länge ≥ 4 eine Sehne enthält, d. h. wenn für jeden einfachen Kreis $C = x_1, x_2, \ldots, x_k, x_1$ mit $k \geq 4$ auch eine Kante $\{x_i, x_j\}$ existiert, so dass x_i und x_j in C nicht aufeinander folgen.

Zeigen Sie, dass jeder Intervall-Graph trianguliert ist.

(b) Ein Graph $G = (V, E)$ heißt *transitiv orientierbar* (oder *Komparabilitätsgraph*), wenn man jeder Kante eine Richtung zuordnen kann, so dass der resultierende gerichtete Graph $G' = (V, F)$ die folgende Bedingung erfüllt:

$$(u, v) \in F \text{ und } (v, w) \in F \Longrightarrow (u, w) \in F \quad \text{für alle } u, v, w \in V.$$

Der Graph $G' = (V, F)$ heißt dann auch eine *transitive Orientierung* von G.

Zeigen Sie, dass der Komplementgraph eines Intervall-Graphen transitiv orientierbar ist. (Dabei entstehe der Komplementgraph eines Graphen durch das Vertauschen von Kanten und Nicht-Kanten.)

(c) Eine *Clique* eines Graphen ist eine Teilmenge von Knoten, die paarweise durch Kanten verbunden sind, eine *maximale Clique* ist eine Clique, die nicht durch die Hinzunahme eines weiteren Knotens zu einer größeren Clique erweitert werden kann.

Zeigen Sie, dass man die maximalen Cliquen eines Intervall-Graphen $G = (V, E)$ so ordnen kann, dass für jeden Knoten $x \in V$ alle diejenigen maximalen Cliquen, die x enthalten, in dieser Ordnung unmittelbar aufeinander folgen.

Hinweis: Verwenden Sie hierfür die Aussagen aus den Aufgabenteilen (a) und (b) sowie das folgende Lemma.

> Sei $G = (V, E)$ ein Intervallgraph, sei $H = (V, F)$ eine transitive Orientierung des Komplementgraphen zu G. Seien A und B zwei maximale Cliquen in G. Dann gibt es eine Kante in F, die einen Endpunkt in A und einen Endpunkt in B besitzt, und alle solche Kanten zwischen A und B haben dieselbe Orientierung in H.

Zeigen Sie zunächst, dass die Orientierung aus F eine lineare Ordnung auf der Menge der maximalen Cliquen definiert. Weisen Sie dann nach, dass diese Ordnung auch die gewünschte Eigenschaft hat, dass alle maximalen Cliquen, die einen Knoten x enthalten, aufeinander folgen.

(d) Zeigen Sie, dass jeder Graph, in dem man die maximalen Cliquen eines Intervall-Graphen $G = (V, E)$ so ordnen kann, dass für jeden Knoten $x \in V$ alle diejenigen maximalen Cliquen, die x enthalten, in dieser Ordnung unmittelbar aufeinander folgen, ein Intervall-Graph ist.

(e) Zeigen Sie, wie man zu einem Graphen G eine binäre Matrix M konstruieren kann, so dass die Matrix M genau dann die Consecutive-Ones-Eigenschaft besitzt, wenn G ein Intervall-Graph ist.

20 Punkte

Aufgabe 22

Zeigen Sie, dass man die Analyse der Approximationsgüte des in der Vorlesung vorgestellten Spannbaum-Algorithmus für das Δ-TSP asymptotisch nicht mehr verbessern kann, indem Sie für unendlich viele natürliche Zahlen n jeweils eine Instanz I_n des Δ-TSP mit mindestens n Knoten angeben, so dass gilt

$$\lim_{n\to\infty} \frac{\mathrm{SB}(I_n)}{\mathrm{Opt}_{\Delta-\mathrm{TSP}}(I_n)} = 2.$$

Dabei seien $\mathrm{SB}(I_n)$ die Kosten der von dem Spannbaum-Algorithmus auf der Instanz I_n berechneten Lösung und $\mathrm{Opt}_{\Delta-\mathrm{TSP}}(I_n)$ die Kosten einer optimalen Lösung. **10 Punkte**

Aufgabe 23

Das *Hamming-TSP* ist der folgende Spezialfall des Traveling-Salesman-Problems:

Eingabe: Ein vollständiger kantengewichteter Graph $G = (V, E, d_H)$, wobei $V = \{v_1, \ldots, v_n\} \subseteq \{0,1\}^m$ eine Menge von Strings der Länge m über dem Alphabet $\Sigma = \{0, 1\}$ ist und für zwei beliebige Knoten u und v die Gewichtung $d_H(\{u, v\})$ der Kante $\{u, v\}$ dem Hamming-Abstand von u und v entspricht.

Zulässige Lösungen: Alle Permutationen von V.

Kosten: Die Kosten einer zulässigen Lösung $(v_{i_1}, \ldots, v_{i_n})$ sind

$$\mathit{cost}(v_{i_1}, \ldots, v_{i_n}) = \left(\sum_{j=1}^{n-1} d_H(\{v_{i_j}, v_{i_{j+1}}\}) \right) + d_H(\{v_{i_n}, v_{i_1}\}).$$

Optimierungsziel: Minimierung.

Zeigen Sie, dass das Hamming-TSP NP-schwer ist.
Hinweis: Geben Sie eine Polynomialzeit-Reduktion von dem Problem des Hamiltonischen Kreises auf die Entscheidungsvariante des Hamming-TSP an. Transformieren Sie hierbei eine Eingabe-Instanz für das Problem des Hamiltonschen Kreises mit n Knoten und m Kanten in eine Instanz des Hamming-TSP mit $n + m$ Knoten aus $\{0,1\}^n$. **10 Punkte**

Abgabe: bis Mittwoch, den 25. Juni 2003, um 14.00 Uhr im Sammelkasten am Lehrstuhl.

Lehrstuhl für Informatik I
Prof. Juraj Hromkovič
Dr. Hans-J. Böckenhauer
Dirk Bongartz

Musterlösung zur 6. Übung zur Vorlesung Algorithmische Grundlagen der Bioinformatik

Aachen, den 30. Juni 2003

Lösung zu Aufgabe 21

(a) Sei G ein Intervall-Graph, der einen einfachen Kreis $[v_0, v_1, v_2, \ldots, v_{l-1}, v_0]$ mit $l > 3$, aber ohne Sehne enthält. Sei weiterhin I_k das Intervall, das im Graph durch v_k dargestellt wird.

Wähle einen Punkt $p_i \in I_{i-1} \cap I_i$ für alle $i = 1, 2, \ldots, l-1$. Da sich I_{i-1} und I_{i+1} nicht überlappen (sonst wäre die Kante $\{v_{i-1}, v_{i+1}\}$ in G vorhanden und damit eine Sehne des Kreises), stellen die p_i eine strikt größer oder strikt kleiner werdende Sequenz dar. Deshalb ist es unmöglich, dass I_0 und I_{l-1} sich überschneiden. Das widerspricht aber der Voraussetzung, dass G eine Kante $\{v_0, v_{l-1}\}$ besitzt.

(b) Sei $\{I_v\}_{v \in V}$ eine Darstellung von $G = (V, E)$ durch Intervalle. Dann sei eine Orientierung F des Komplement-Graphen $\overline{G} = (V, \overline{E})$ wie folgt definiert:

$$(x, y) \in F \Leftrightarrow I_x < I_y \quad (\text{für alle } \{x, y\} \in \overline{E}).$$

Dabei bedeutet $I_x < I_y$, dass das Intervall I_x vollständig zur Linken des Intervalls I_y liegt. (I_x und I_y sind disjunkt, da $\{x, y\}$ sonst eine Kante in G wäre.) Dann ist $\overline{G}$ sicherlich transitiv orientiert, da $I_x < I_y < I_z$ impliziert, dass $I_x < I_z$ gilt.

(c) Sei $G = (V, E)$ ein trianguliert Graph und sei $H = (V, F)$ eine transitive Orientierung des Komplement-Graphen $\overline{G} = (V, \overline{E})$. A_1 und A_2 seien maximale Cliquen in G.

Dann haben nach dem Lemma alle Kanten, die einen Endpunkt in A_1 und einen Endpunkt in A_2 haben, dieselbe Orientierung in H. Ordne nun die maximalen Cliquen gemäß der Richtung der Kanten in H. Damit gilt $A_1 < A_2$ genau dann, wenn es eine Kante aus H gibt, die A_1 und A_2 verbindet und in Richtung A_2 orientiert ist. Da gemäß des Lemmas zwischen je zwei Cliquen eine Kante existiert, haben wir hiermit für *jedes* Paar von Cliquen eine Ordnung festgelegt. Wir zeigen im Folgenden, dass die so definierte Ordnung transitiv ist. Damit folgt dann, dass es sich um eine lineare Ordnung auf der Menge der maximalen Cliquen handelt.

Es gelte, dass $A_1 < A_2$ und $A_2 < A_3$. Dann gibt es Kanten (a_1, a_2') und (a_2'', a_3) in H mit $a_1 \in A_1$, $a_2', a_2'' \in A_2$ und $a_3 \in A_3$. Wenn entweder $\{a_2', a_3\} \notin E$ oder $\{a_1, a_2''\} \notin E$, dann gilt $(a_1, a_3) \in F$ und damit $A_1 < A_3$. Deshalb nehmen wir an, dass die Kanten $\{a_1, a_2''\}$ und $\{a_2', a_3\}$ in E liegen. Weil A_2 eine Clique ist, liegt auch $\{a_2'', a_2'\}$ in E. Da wir aus (a) wissen, dass G keinen einfachen Kreis der Länge ≥ 4 ohne Sehne besitzt, ist $\{a_1, a_3\} \notin E$, also gilt $(a_1, a_3) \in F$.

Sei $A_1, \ldots, A_m$ die so definierte lineare Anordnung der maximalen Cliquen. Angenommen, es existieren Cliquen $A_i < A_j < A_k$ mit $x \in A_i$, $x \notin A_j$ und $x \in A_k$. Da $x \notin A_j$, gibt es einen Knoten $y \in A_j$, so dass $\{x, y\} \notin E$. Aber $A_i < A_j$ impliziert, dass $(x, y) \in F$, wohingegen $A_j < A_k$ bedeutet, dass $(y, x) \in F$. Dies ist ein Widerspruch, also definiert die Orientierung aus F eine lineare Ordnung auf der Menge der maximalen Cliquen und alle maximalen Cliquen, die einen Knoten x enthalten folgen in dieser Ordnung unmittelbar aufeinander.

(d) Sei $A_1, \ldots, A_m$ die wie in (c) definierte lineare Anordnung der maximalen Cliquen von G. Für jeden Knoten x sei $I(x)$ die Menge der Indizes aller maximalen Cliquen aus G, die x enthalten. Wie in (c) gezeigt, folgen für jeden Knoten $x \in V$ die Indizes in $I(x)$ unmittelbar aufeinander, bilden also ein Intervall auf der Menge $\{1, \ldots, m\}$ der Indizes. Damit können wir jeden Knoten x mit dessen Intervall $I(x)$ identifizieren. Zwei solche Intervalle haben einen nichtleeren Schnitt genau dann, wenn es eine Clique gibt, die beide zugehörigen Knoten enthält. Damit sind diese Knoten durch eine Kante miteinander verbunden, G ist also ein Intervall-Graph.

(e) Sei $n = |V|$ die Anzahl der Knoten des Graphen G und seien $A_1, A_2, \ldots, A_m$ die maximalen Cliquen von G. Dann ist die Clique-Matrix M eine $(n \times m)$-Matrix, mit $M(i,j) = 1$, falls $v_i \in A_j$ und $M(i,j) = 0$ sonst.

Falls G ein Intervall-Graph ist, dann können wir gemäß (c) die Ordnung der Cliquen so wählen, dass alle maximalen Cliquen, die einen bestimmten Knoten enthalten, aufeinander folgen. Wenn wir nun die Spalten der Clique-Matrix entsprechend dieser Ordnung permutieren, dann folgen in jeder Zeile von M alle Einsen unmittelbar aufeinander. Also besitzt die Matrix die Consecutive-Ones-Eigenschaft.

Falls die Clique-Matrix die Consecutive-Ones Eigenschaft besitzt, dann gibt es eine Anordnung der Spalten, so dass in jeder Zeile von M alle Einsen unmittelbar aufeinander folgen. Damit ist eine Ordnung der Cliquen definiert, in der für jeden Knoten x alle maximalen Cliquen, die x enthalten, unmittelbar aufeinander folgen. Nach (d) ist G also ein Intervall-Graph.

Lösung zu Aufgabe 22

Sei n eine positive, gerade Zahl. Betrachte dann als Eingabeinstanz des Δ-TSP den vollständigen Graphen $I_n = (V, E)$ mit n Knoten aus Abbildung 1(a), und die Gewichtsfunktion $c : E \to \{1, 1+\varepsilon, 2\}$. Die durchgezogenen Kanten haben Kantengewicht 1, die gepunkteten Kanten (diese verbinden alle „Außenknoten“ der jeweils linken und rechten Seite des Graphen) Kantengewicht 2, und die gestrichelten Kanten (diese verbinden die „Außenknoten“ einer Seite mit allen Knoten der gegenüberliegenden Seite) besitzen jeweils ein Kantengewicht von $1+\varepsilon$, wobei $\varepsilon > 0$ einen kleinen, positiven Wert bezeichnet (es ist beispielweise ausreichend, $\varepsilon := \frac{1}{n^2}$ zu wählen). Man beachte, dass die beiden zuletzt genannten Kantenarten aus Gründen der Übersichtlichkeit jeweils nur teilweise eingezeichnet sind.
Offensichtlich ist der in Abbildung 1(b) dargestellte Baum der einzige minimale Spannbaum von I_n bezüglich der Gewichtsfunktion c, und es gilt $cost(T) = n - 1$. Die Ausgabe des Algorithmus SB ist – beginnend mit dem Knoten in der linken oberen Ecke – beispielsweise der in Abbildung 1(c) gezeigte Hamiltonkreis $\overline{H}$ mit den Kosten $cost(\mathrm{SB}(I_n)) = cost(\overline{H}) = 4 + 2 \cdot (1+\varepsilon) + (n-6) \cdot 2$.
Man beachte hierbei, dass jeder mögliche von SB berechnete Hamiltonkreis, unabhängig von der Reihenfolge, in der der Algorithmus die Knoten während der Tiefensuche besucht, Kosten von mindestens $4 + 2 \cdot (1+\varepsilon) + (n-6) \cdot 2$ hat.
Ein optimaler Hamiltonkreis H_{Opt} ist jedoch beispielsweise der in Abbildung 1(d) gezeigte mit den Kosten $cost(Opt_{\Delta-\mathrm{TSP}}(I_n)) \leq n \cdot (1+\varepsilon)$. Dann

$$\frac{\mathrm{SB}(I_n)}{Opt_{\Delta-\mathrm{TSP}}(I_n)} = \frac{cost(\overline{H})}{cost(H_{\mathrm{Opt}})} = \frac{2n - 6 + 2 \cdot \varepsilon}{n + n \cdot \varepsilon},$$

was mit wachsendem n und beispielsweise $\frac{1}{n^2}$ als Wahl für ε gegen den Wert 2 strebt, d.h.

$$\lim_{n \to \infty} \frac{\mathrm{SB}(I_n)}{Opt_{\Delta-\mathrm{TSP}}(I_n)} = 2.$$

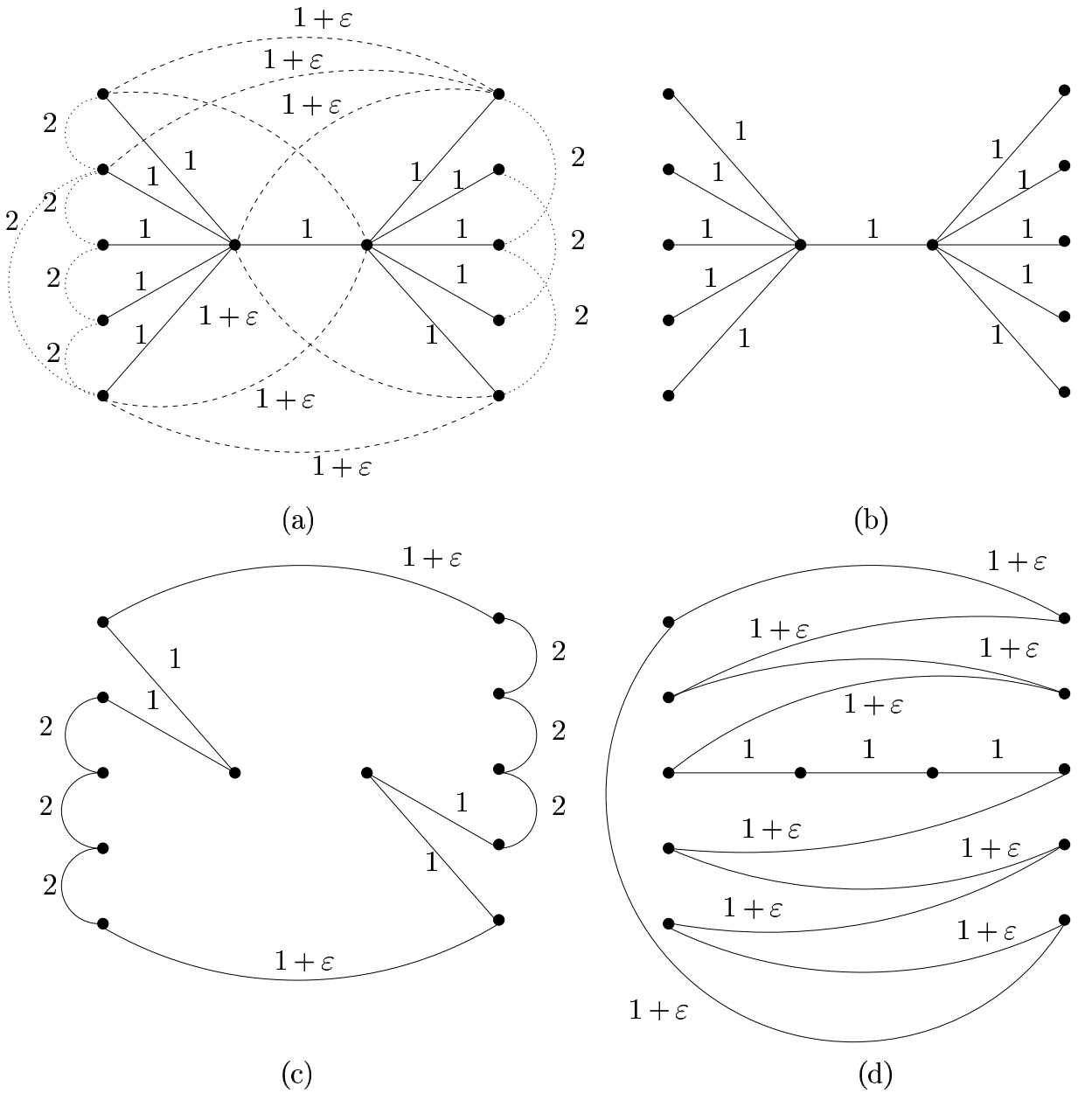


Abbildung 1: Die Konstruktion in Aufgabe 22.

Falls die Kantenkosten, wie in der Vorlesung definiert, auf natürliche Zahlen beschränkt sind, dann kann man eine entsprechende Folge von Beispielinstanzen dadurch konstruieren, indem man die Kosten der obigen Instanz mit einem geeigneten Faktor multipliziert.

Lösung zu Aufgabe 23

Bemerkung: Der Hinweis zur Lösung dieser Aufgabe war leider irreführend, die Aufgabe wird daher aus der Wertung genommen. Für sinnvolle Bearbeitungen der Aufgabe werden Zusatzpunkte vergeben.

Wir geben im Folgenden einen Beweis an, der auf einer anderen Reduktion basiert.

Das L_1-TSP ist die Variante des TSP, bei der die Knoten Punkten in der Ebene (mit positiven Koordinaten) entsprechen und die Kantenkosten sich als Abstand zwischen den Knoten in der L_1-Norm ergeben, zum Beispiel haben die Knoten mit den Koordinaten (a, b) und (c, d) den Abstand $|a - c| + |b - d|$.

Von dem L_1-TSP ist bekannt, dass es stark NP-schwer ist, d. h. auch bei Einschränkung des Problems auf solche Eingabe-Instanzen, bei denen der größte in einer Koordinate auftretende Wert polynomiell in der Länge der (binär kodierten) Eingabe ist, ist noch NP-schwer. (Man beachte, dass wegen der Binärkodierung der Eingabe im Allgemeinen die größten Werte der

Koordinaten sehr wohl exponentiell groß in der Länge der Eingabe sein können.) Wir nennen den Spezialfall des L_1-TSP mit derart eingeschränkten Eingaben im Folgenden Pol-L_1-TSP und geben eine Polynomialzeit-Reduktion von dem Pol-L_1-TSP auf das Hamming-TSP an.
Diese Reduktion basiert auf der Idee, die Koordinaten der Knoten in der gegebenen Pol-L_1-TSP-Instanz *unär* zu kodieren, dann entspricht der L_1-Abstand zwischen den Knoten genau dem Hamming-Abstand der Kodierungen. Formal lässt sich diese Reduktion wie folgt beschreiben: Sei eine Menge von Punkten in der Ebene $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ als Eingabe für das Pol-L_1-TSP gegeben, sei k der Wert der größten auftretenden Koordinate. Wir definieren hierfür eine Eingabe für das Hamming-TSP wie folgt: Sei $G = (V, E, d_H)$ ein vollständiger Graph mit $V = \{v_1, \ldots, v_n\} \subseteq \{0,1\}^{2k}$, wobei für alle $1 \leq i \leq n$ gelte

$$v_i = 0^{k-x_i}1^{x_i}0^{k-y_i}1^{y_i}.$$

Weiter sei $d_H(v_i, v_j)$ der Hamming-Abstand der Knoten v_i und v_j für alle $1 \leq i, j \leq n$.
Dann gilt offenbar, dass $d_H(v_i, v_j) = |x_i - x_j| + |y_i - y_j|$. Also haben wir das gegebene Problem nur umformuliert und es gilt offenbar, dass jede Lösung für die konstruierte Hamming-TSP-Instanz einer Lösung für die gegebene Pol-L_1-TSP-Instanz mit den gleichen Kosten entspricht und umgekehrt.
Da wir vorausgesetzt haben, dass die Werte aller Koordinaten der Knoten in der Eingabe für das Pol-L_1-TSP nur polynomiell groß sind, sind auch ihre Unär-Kodierungen nur polynomiell groß, also ist die beschriebene Reduktion auch in polynomieller Zeit durchführbar.

Lehrstuhl für Informatik I
Prof. Juraj Hromkovič
Dr. Hans-J. Böckenhauer
Dirk Bongartz

7. Übung zur Vorlesung Algorithmische Grundlagen der Bioinformatik

Aachen, den 2. Juli 2003

Aufgabe 24

Für den Greedy-Superstring-Algorithmus für das SCS sind wir davon ausgegangen, dass die Eingabe eine teilstringfreie Menge von Strings ist. Zeigen Sie, dass die Teilstring-Freiheit auch für alle Mengen von Strings, die der Greedy-Superstring-Algorithmus als Zwischenergebnisse konstruiert, erhalten bleibt.
Sei hierfür eine Menge $S = \{s_1, \ldots, s_n\}$ von Strings über einem Alphabet Σ als Eingabe gegeben und sei S_i die Menge S nach i ausgeführten Schritten des Greedy-Superstring-Algorithmus. Für jeden String $s \in S_i$ gilt entweder $s = s_j$ für ein $j \in \{1, \ldots, n\}$ oder $s = \langle s_{j_1}, \ldots, s_{j_k} \rangle$. Wir definieren $\mathit{first}(s) := s_{j_1}$ [bzw. $\mathit{first}(s) := s_j$] und $\mathit{last}(s) := s_{j_k}$ [bzw. $\mathit{last}(s) := s_j$].
Zeigen Sie, dass für zwei beliebige Strings s und t in S_i gilt, dass weder $\mathit{first}(s)$ noch $\mathit{last}(s)$ ein Teilstring von t ist. **10 Punkte**

Aufgabe 25

Wir betrachten eine Variante des Greedy-Superstring-Algorithmus, bei der die Verwendung von Overlaps eines Strings mit sich selbst erlaubt ist, den so genannten TGreedy-Algorithmus.

Eingabe: Eine teilstringfreie Menge von Strings $S = \{s_1, \ldots, s_n\}$.

1. Setze $T := \emptyset$.
2. **while** $|S| > 0$ **do**
   - 2.1 Bestimme $s, t \in S$ mit dem maximalen Overlap aller Strings in S.
   - 2.2 Falls $s \neq t$, dann lösche s und t aus S und füge $\langle s, t \rangle$ hinzu.
   - 2.3 Falls $s = t$, dann lösche s aus S und füge s zu T hinzu.
3. Wende den Greedy-Superstring-Algorithmus auf die Menge T an.

Ausgabe: Der von dem Greedy-Superstring-Algorithmus auf T berechnete String w_{tgreedy}.

Man kann zeigen, dass der TGreedy-Algorithmus ein 3-Approximationsalgorithmus für das SCS ist. Wir wollen im Folgenden zeigen, dass der TGreedy-Algorithmus und der Greedy-Superstring-Algorithmus bezüglich ihrer Approximation unvergleichbar sind.

(a) Geben Sie ein Beispiel an, auf dem der Greedy-Superstring-Algorithmus eine bessere Approximation liefert als der TGreedy-Algorithmus.

(b) Geben Sie ein Beispiel an, auf dem der TGreedy-Algorithmus eine bessere Approximation liefert als der Greedy-Superstring-Algorithmus.

10 Punkte

(bitte wenden)

Aufgabe 26

Betrachten Sie die Einschränkung des SCS auf solche Eingabe-Instanzen S, für deren maximale Kompression gilt

$$comp(S) \leq \frac{2}{3} \cdot ||S||.$$

Zeigen Sie, das der Greedy-Superstring-Algorithmus ein 2-Approximationsalgorithmus für das SCS auf solchen Eingabe-Instanzen ist. **10 Punkte**

Aufgabe 27

Sei $S = \{s_1, \ldots, s_n\}$ eine teilstringfreie Menge von Strings, sei $t \notin S$ ein weiterer String, so dass auch $S \cup \{t\}$ teilstringfrei ist. Mit $greedy(T)$ bezeichnen wir die Länge eines von dem Greedy-Superstring-Algorithmus berechneten Superstrings einer Menge T von Strings.
Beweisen oder widerlegen Sie die folgende Aussage:

$$greedy(S \cup \{t\}) \leq greedy(S) + |t|.$$

10 Punkte

Abgabe: bis Mittwoch, den 9. Juli 2003, um 14.00 Uhr im Sammelkasten am Lehrstuhl.

Lehrstuhl für Informatik I
Prof. Juraj Hromkovič
Dr. Hans-J. Böckenhauer
Dirk Bongartz

Musterlösung zur 7. Übung zur Vorlesung Algorithmische Grundlagen der Bioinformatik

Aachen, den 11. Juni 2003

Lösung zu Aufgabe 24

Wir zeigen im Folgenden, dass für zwei beliebige Strings s und t in S_i gilt, dass weder $\mathit{first}(s)$ noch $\mathit{last}(s)$ ein Teilstring von t ist.
Zu Beginn des Algorithmus gilt für alle $s \in S$, dass $\mathit{first}(s) = \mathit{last}(s) = s$, so dass die Behauptung unmittelbar aus der Voraussetzung folgt, das S teilstringfrei ist.
Nehmen wir nun an, dass die Behauptung durch irgendeinen Merge zweier Strings t_1 und t_2 zu $t = \langle t_1, t_2 \rangle$ verletzt wird, und es gelte o.B.d.A. $\mathit{first}(s)$ ist ein Teilstring von t. In diesem Fall ließe sich t darstellen als $t = u\ \mathit{first}(s)\ v$ für irgendwelche $u, v \in \Sigma^*$. Da $\mathit{first}(s)$ weder Teilstring von t_1 noch von t_2 ist, muss der Overlap $Ov(t_1, t_2)$ der beiden Strings t_1 und t_2 echt in $\mathit{first}(s)$ enthalten sein. Daher gelten die beiden Ungleichungen $|\mathit{first}(s)| > ov(t_1, t_2)$ und $|u| < pref(t_1, t_2)$ und folglich auch $ov(t_1, s) > ov(t_1, t_2)$. In diesem Fall wären durch den Algorithmus Greedy-Superstring jedoch nicht die beiden Strings t_1 und t_2 verschmolzen worden, sondern der Algorithmus hätte die beiden Strings t_1 und s verknüpft. Dies ist ein Widerspruch zu der Annahme, die Verletzung wäre durch einen Greedy-Merge von t_1 und t_2 zustande gekommen.

Lösung zu Aufgabe 25

(a) Wir betrachten die Menge $S = \{c(ab)^k, (ab)^{k+1}a, (ba)^k c\}$ von Strings über dem Alphabet $\Sigma = \{a, b, c\}$. Der in der Vorlesung vorgestellte Algorithmus Greedy-Superstring bildet als kürzesten gemeinsamen Superstring eindeutigerweise den String $c(ab)^{k+1}ac$ der Länge $2k + 5$, wohingegen der TGreedy-Algorithmus zunächst den String $(ab)^{k+1}a$ in die Menge T verschiebt, da die Überlappung dieses Strings mit sich selbst die größte Überlappung in der Menge S bildet. Dann fügt TGreedy die beiden verbleibenden Strings aus S zusammen und terminiert schließlich mit der Ausgabe $c(ab)^k ac(ab)^{k+1}a$ der Länge $4k + 6$.

Dieses Beispiel zeigt, dass der TGreedy-Algorithmus sich nicht auf allen Eingaben besser verhält als der (einfachere) Greedy-Superstring-Algorithmus.

(b) Das Ergebnis aus Aufgabenteil (a) ließe die Vermutung zu, dass der einfache Greedy-Superstring-Algorithmus stets bessere Ergebnisse liefert als der Algorithmus TGreedy, obwohl wir keine bessere obere Schranke für die Approximationsgüte des Greedy-Superstring-Algorithmus angeben können als 4. Dass diese Vermutung jedoch falsch ist, zeigt das folgende Gegenbeispiel.

Betrachte dazu die Menge $S = \{cab^k, ab^k ab^k a, b^k dab^{k-1}\}$ von Strings über dem Alphabet $\Sigma = \{a, b, c, d\}$. Wie in Aufgabenteil (a) findet TGreedy als längsten Overlap der Strings aus S den Overlap von $ab^k ab^k a$ mit sich selbst und „separiert“ diesen String in die Menge T. Daraufhin führt der Algorithmus zunächst einen Merge der beiden Strings $s = cab^k$ und $s' = b^k dab^{k-1}$ zu $\langle s, s' \rangle = cab^k dab^{k-1}$ durch und generiert in Schritt 3 den Superstring $cab^k dab^k ab^k a$ der Länge $3k + 6$. Greedy-Superstring hingegen verbindet zunächst die ersten beiden Strings aus S zu $cab^k ab^k a$ (Overlap der Länge $k+1$), was unmittelbar dazu führt, dass der letzte String nur noch vollständig konkateniert werden kann, was zu dem Ergebnis $cab^k ab^k ab^k dab^{k-1}$ der Länge $4k + 5$ führt.

In diesem Fall führt der TGreedy-Algorithmus also zu einem besseren Ergebnis als der Algorithmus Greedy-Superstring.

Lösung zu Aufgabe 26

Wir zeigen im Folgenden, dass für den Fall

$$comp(S) \leq \frac{2}{3} \cdot ||S|| \tag{1}$$

der Greedy-Superstring-Algorithmus ein 2-Approximationsalgorithmus für das SCS ist. Dabei bezeichne $comp(S)$ die Kompression einer optimalen Lösung des MCCS für dei Eingabe S.
Für den Beweis verwenden wir unter anderem das Resultat des Satzes 8.5 der Vorlesung, dass Greedy-Superstring einen 2-Approximationsalgorithmus für das MCCS darstellt, d.h. dass gilt:

$$\frac{comp(w_{opt})}{comp(w_{greedy})} \leq 2, \tag{2}$$

wobei w_{greedy} die durch den Greedy-Superstring-Algorithmus ausgegebene Lösung bezeichne und w_{opt} ein optimaler Superstring für die Eingabeinstanz $S = \{s_1, \ldots, s_n\}$ sei.
Man beachte darüber hinaus, dass für die Kompression $comp(w)$ eines Wortes w gilt:

$$comp(w) = ||S|| - |w|. \tag{3}$$

Gleichung (1) lässt sich wie zunächst mit Hilfe von Gleichung (3) wie folgt umformen:

$$\begin{aligned} & comp(S) \leq \frac{2}{3} \cdot ||S|| \\ \iff \quad & ||S|| - |w_{opt}| \leq \frac{2}{3} \cdot ||S|| \\ \iff \quad & 3 \cdot |w_{opt}| \geq ||S|| \end{aligned} \tag{4}$$

Ausgehend von Gleichung (2) läßt sich dann die Approximationsgüte des Algorithmus Greedy-Superstring wie folgt abschätzen:

$$\begin{aligned} & \frac{comp(w_{opt})}{comp(w_{greedy})} \leq 2 \\ \overset{(3)}{\Longrightarrow} \quad & \frac{||S|| - |w_{opt}|}{||S|| - |w_{greedy}|} \leq 2 \\ \Longrightarrow \quad & ||S|| - |w_{opt}| \leq 2 \cdot ||S|| - 2 \cdot |w_{greedy}| \\ \Longrightarrow \quad & 2 \cdot |w_{greedy}| \leq ||S|| + |w_{opt}| \\ \overset{(4)}{\Longrightarrow} \quad & 2 \cdot |w_{greedy}| \leq 4 \cdot |w_{opt}| \\ \Longrightarrow \quad & \frac{|w_{greedy}|}{|w_{opt}|} \leq 2 \end{aligned}$$

Lösung zu Aufgabe 27

Wir zeigen im Folgenden, dass die Aussage

$$greedy(S \cup \{t\}) \leq greedy(S) + |t|$$

für eine Menge $S = \{s_1, \ldots, s_n\}$ teilstringfreier Strings und einen String $t \notin S$ im Allgemeinen falsch ist.

Hierzu betrachten wir die Menge $S = \{ca^m, a^{m+1}c^m, c^m b^{m+1}, b^m c\}$ und den String $t = b^{m+1}a^{m+1}$. Dann ist die Menge S offensichtlich teilstringfrei und auch t ist nicht Teilstring eines der Strings aus S oder umgekehrt.
Wenden wir nun den GREEDY-SUPERSTRING-Algorithmus auf die Menge S an, so terminiert dieser mit der eindeutigen Ausgabe $ca^{m+1}c^m b^{m+1}c$, und es gilt:

$$\begin{aligned} greedy(S) + |t| &= |ca^{m+1}c^m b^{m+1}c| + |b^{m+1}a^{m+1}| \\ &= (3m+4) + (2m+2) \\ &= 5m+6 \end{aligned}$$

GREEDY-SUPERSTRING angewendet auf $S \cup \{t\}$ liefert hingegen die Ausgabe $b^m c^m b^{m+1} a^{m+1} c^m a^m$ mit Länge $6m+2$. Es gilt also offensichtlich

$$greedy(S \cup \{t\}) = 6m+2 > 5m+6 = greedy(S) + |t| \quad \text{für alle } m \geq 5,$$

so dass die ursprüngliche Aussage falsch ist.

Lehrstuhl für Informatik I
Prof. Juraj Hromkovič
Dr. Hans-J. Böckenhauer
Dirk Bongartz

8. Übung zur Vorlesung Algorithmische Grundlagen der Bioinformatik

Aachen, den 9. Juli 2003

Aufgabe 28

Sei $S = \{s_1, \ldots, s_n\}$ ein Spektrum mit Strings der Länge l über einem Alphabet Σ. In der Vorlesung haben wir gesehen, dass jeder einfach-kompatible String zu S einem Eulerpfad in dem Spektrum-Graphen entspricht.

(a) Geben Sie ein Beispiel für ein Spektrum an, zu dem ein kompatibler String, aber kein einfach-kompatibler String existiert.

(b) Geben Sie einen Algorithmus an, der zu einem gegebenen Spektrum S einen kompatiblen String ausgibt, falls ein solcher existiert, und eine Fehlermeldung sonst.

10 Punkte

Aufgabe 29

Geben Sie einen effizienten Algorithmus an, der für einen gegebenen String s und eine gegebene natürliche Zahl l alle Teilstrings von s der Länge l zusammen mit ihrer Häufigkeit in s ausgibt.

10 Punkte

Aufgabe 30

Sei X_i eine Zufallsvariable mit Wertebereich $\{0, 1\}$ für $1 \leq i \leq m$, sei $X = \sum_{i=1}^{m} X_i$. Zeigen Sie die folgende Aussage über die Varianz von X:

$$Var[X] = \sum_{1 \leq i,j \leq m} (E[X_i \cdot X_j] - E[X_i] \cdot E[X_j]).$$

10 Punkte

Aufgabe 31

In der Vorlesung haben wir mit dem Viterbi-Algorithmus ein Verfahren kennengelernt, um für ein HMM und einen String den wahrscheinlichsten Pfad zu bestimmen, der diesen String generiert. Wir wollen nun eine ähnliche Fragestellung untersuchen: Zu einem gegebenen HMM und einem String wollen wir die Wahrscheinlichkeit bestimmen, mit der das HMM diesen String generiert.
Geben Sie einen effizienten Algorithmus an, der dieses Problem löst.
Hinweis: Verwenden Sie eine geeignete Modifikation des Viterbi-Algorithmus. **10 Punkte**

Abgabe: bis Mittwoch, den 16. Juli 2003, um 14.00 Uhr im Sammelkasten am Lehrstuhl.

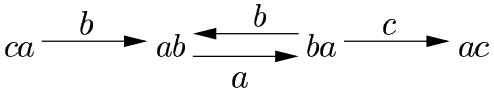
Lehrstuhl für Informatik I
Prof. Juraj Hromkovič
Dr. Hans-J. Böckenhauer
Dirk Bongartz

Musterlösung zur 8. Übung zur Vorlesung Algorithmische Grundlagen der Bioinformatik

Aachen, den 23. Juni 2003

Lösung zu Aufgabe 28

(a) Ein Beispiel für ein Spektrum, zu dem ein kompatibler, aber kein einfach kompatibler String existiert, ist das Spektrum $S = \{cab, aba, bab, bac\}$. Der Spektrum-Graph zu diesem Spektrum sieht wie folgt aus:



Es ist offensichtlich, dass dieser Graph keinen Eulerschen Pfad enthält. Deshalb existiert kein einfach kompatibler String zum Spektrum S. Allerdings ist ein kompatibler String durch den String $s = cababac$ gegeben. Dieser entspricht einem Pfad durch den Spektrum-Graphen, der die Kante $\{ab, ba\}$ zweimal durchläuft. Das Problem entsteht hier dadurch, dass wir nur das Spektrum gegeben haben, und nicht wissen, in welcher Häufigkeit die einzelnen Strings des Spektrums in einem kompatiblen String vorkommen.

(b) Ein kompatibler String lässt sich gemäß folgender Idee konstruieren: Man konstruiert zunächst den Spektrum-Graphen G zum gegebenen Spektrum S. Darin sucht man nun einen Pfad, in dem jede Kante von G mindestens einmal besucht wird. Durch diesen Pfad wird dann ein zu S kompatibler String s dargestellt.

Um nun zu bestimmen, ob es einen solchen Pfad in G gibt, bestimmt man die (maximalen) starken Zusammenhangskomponenten $S_1, \ldots, S_k$ von G. (Eine starke Zusammenhangskomponente (*strongly connected component*, SCC) ist ein induzierter Teilgraph, in dem zwischen je zwei beliebigen Knoten ein (gerichteter) Pfad existiert. Die starken Zusammenhangskomponenten eines gerichteten Graphen lassen sich in polynomieller Zeit bestimmen, vgl. hierfür z. B. das Buch von Cormen, Leiserson, Rivest und Stein.) Wir betrachten nun den Multigraphen G', der sich ergibt, wenn man die SCCs in G jeweils zu einem einzelnen Knoten kontrahiert.

Man kann leicht zeigen, dass der Graph G genau dann einen Pfad besitzt, der jede Kante mindestens einmal besucht, wenn der SCC-Graph G' ein Pfad ist, also keine Knoten mit Eingangsgrad ≥ 2 oder mit Ausgangsgrad ≥ 2 enthält:

Offenbar kann G' wegen der Maximalität der SCCs keinen Kreis enthalten. In einer SCC können wir immer einen Pfad finden, der in einem beliebigen Knoten beginnt, in einem beliebigen Knoten endet und alle Kanten innerhalb der SCC jeweils mindestens einmal enthält, da jeder Knoten (und somit auch jede Kante) von jedem Knoten aus erreichbar ist. Wenn der SCC-Graph ein Pfad ist, dann können wir also alle SCCs entlang dieses Pfades besuchen und komplett durchlaufen. Es bleibt zu zeigen, dass dies nicht mehr möglich ist, wenn eine SCC einen Eingangs- oder Ausgangsgrad von ≥ 2 hat. Wir nehmen an, die SCC S_i habe zwei eingehende Kanten e_1 und e_2. Ein Pfad P durch G, der jede Kante mindestens einmal besucht, erreiche S_i zum ersten Mal über die Kante e_1. Um auch die Kante e_2 zu durchlaufen, muss P die SCC S_i wieder verlassen. Da der SCC-Graph G' aber keinen Kreis enthält, kann P nicht wieder zu S_i zurückkehren, wenn er diese Komponente einmal verlassen hat, also auch nicht über die Kante e_2. Also kann P die

Kante e_2 nicht besuchen. Eine ähnliche Argumentation zeigt, dass es auch in dem Fall, dass eine Komponente zwei ausgehende Kanten in G' hat, keinen Pfad gibt, der beide diese Kanten besucht.

Der Algorithmus zur Bestimmung eines kompatiblen Strings lässt sich also wie folgt zusammenfassen:

Eingabe: Ein Spektrum S.

(a) Konstruiere den Spektrum-Graphen $G = (V, E)$ zu S.

(b) Bestimme die (maximalen) starken Zusammenhangskomponenten $S_1, \ldots, S_k$ mit $S_i = (V_i, E_i)$ und bestimme den SCC-Multigraphen G' durch Kontraktion jeder SCC zu einem einzelnen Knoten.

(c) **if** es gibt einen Knoten in G' mit Eingangs- oder Ausgangsgrad ≥ 2 **then**
return 'Es gibt keinen kompatiblen String zum eingegebenen Spektrum'
else
Bestimme Pfad P, der alle Kanten aus G' enthält (da G zusammenhängend ist und G' keinen Knoten mit Eingangs- oder Ausgangsgrad ≥ 2 hat, muss G' ein Pfad sein)

(d) **for all** $1 \leq i \leq k$ **do**
$P_i := \emptyset$;
Wähle einen beliebigen Knoten $x \in V_i$
for all $e = (v, w) \in E_i$ **do**
Suche einen Pfad p von x nach v, über die Kante e und wieder zurück zu x.
$P_i := (P_i, p)$; (Hänge P_i und p hintereinander)
{Verbesserung: Hierbei schon besuchte zusätzliche Kanten müssen nicht noch gezielt angesteuert werden.}

(e) Bilde den gesamten Pfad E durch G, der alle Kanten von G mindestens einmal enthält aus P und $P_1, \ldots, P_k$:
Seien v_i und z_i die Knoten, die mit den Kanten inzidieren, die S_i mit der in P vorhergehenden, bzw. nachfolgenden Zusammenhangskomponente verbinden. Erweitere P_i, $1 \leq i \leq k$ zu P_i' derart, dass $P_i' = v_i, \ldots, P_i, \ldots, z_i$. Sei o.B.d.A. $P = S_1, \ldots, S_k$ dann ist $E = P_1', \ldots, P_k'$.

(f) Bestimme den String s, der dem Pfad E im Spektrumgraphen G entspricht.

Ausgabe: Der zu S kompatible String s.

Lösung zu Aufgabe 29

Der folgende Algorithmus gibt zu einem gegebenen String s und einer gegebenen Zahl l alle Teilstrings von s der Länge l zusammen mit ihrer Häufigkeit aus:

Eingabe: Ein String $s = s_1 \ldots s_n, n \in \mathbb{N}$ und eine natürliche Zahl l.

1. Sei T ein (zunächst leerer) binärer Suchbaum. Ein Schlüssel k ist ein gefundener Teilstring von s, der alphabetisch im Baum einsortiert ist. Die Werte $v(k)$ sind die Häufigkeiten, mit denen die Teilstrings in s auftreten.

2. **for** $i = 1$ **to** $n - l + 1$ **do**
$t := s_i \ldots s_{i+l-1}$
Suche t in T;
if t ist in T enthalten **then**
$v(t) := v(t) + 1$
else

Füge den Schlüssel t in den Baum ein.
$v(t) = 1$

3. Durchlaufe T und gebe jeden Schlüssel mit seinem Wert aus.

Für diesen Algorithmus ergibt sich folgende Laufzeitabschätzung: In dem String s gibt es $n-l+1$ mögliche Anfangspositionen für einen Teilstring der Länge l, also gibt es auch höchstens $n-l+1$ verschiedene Strings der Länge l, die in s vorkommen. Schritt 2 hat somit eine Laufzeit in $O((n-l+1)\cdot\log(n-l+1)) \in O(n\log n)$. Schritt 3 läuft in $O(n-l+1) \in O(n)$. Insgesamt ergibt sich somit eine Laufzeit von $O(n\cdot\log n)$.

Lösung zu Aufgabe 30

Die Behauptung folgt unmittelbar aus der Linearität des Erwartungswerts:

$$\begin{aligned}
Var[X] &= E[X^2] - E[X]^2 \\
&= E\left[\sum_{i=1}^{m}\sum_{j=1}^{m} X_i \cdot X_j\right] - E\left[\left(\sum_{i=1}^{m} X_i\right)\cdot\left(\sum_{j=1}^{m} X_j\right)\right] \\
&= \sum_{i=1}^{m}\sum_{j=1}^{m} E[X_i \cdot X_j] - (\sum_{i=1}^{m} E[X_i]\cdot\sum_{j=1}^{m} E[X_j]) \\
&= \sum_{1\le i,j\le m} E[X_i \cdot X_j] - \sum_{1\le i,j\le m} E[X_i]\cdot E[X_j] \\
&= \sum_{1\le i,j\le m} (E[X_i \cdot X_j] - E[X_i]\cdot E[X_j])
\end{aligned}$$

Lösung zu Aufgabe 31

Beim naiven Ansatz zur Bestimmung der Wahrscheinlichkeit, mit der ein HMM mit m Zuständen den String $x = x_1 \ldots x_n$ generiert, würde man die Wahrscheinlichkeit für jeden möglichen Pfad, durch den x generiert wird, berechnen, und diese Wahrscheinlichkeiten aufsummieren. Da dieser naive Ansatz allerdings die Wahrscheinlichkeiten von m^n Pfaden berechnen muss und somit exponentielle Laufzeit hat, ist er nicht praktikabel.
Eine effiziente Lösung für das Problem benutzt daher eine geeignete Modifikation des Viterbi-Algorithmus. Dabei stellt $f_p(i) = Prob[x_1 \ldots x_i|p]$ die bedingte Wahrscheinlichkeit dar, dass $x_1 \ldots x_i$ generiert wurde, wenn sich das HMM in Zustand p befindet, also die Wahrscheinlichkeit, dass das Präfix $x_1 \ldots x_i$ von x auf einem Pfad generiert wurde, der in dem Zustand p endet. Der folgende Algorithmus (in der Literatur als *Forward-Algorithmus* bezeichnet) berechnet mit Hilfe dynamischer Programmierung alle Werte $f_p(i)$:

Eingabe: Ein HMM $\mathcal{M} = \{\Sigma, Q, q_0, \delta, \eta\}$ und ein String $X = x_1 \ldots x_n \in \Sigma^*$.

1. Initialisierung:
   $f_{q_0}(0) := 1$
   for all $q \in Q - \{q_0\}$ **do**
   $f_q(0) := 0$

2. Berechnung der $f_q(i)$:
   for $i = 1$ **to** n **do**
   for all $q \in Q - \{q_0\}$ **do**
   $f_q(i) := \eta(q, x_i) \cdot \sum_{p\in Q} f_p(i-1) \cdot \delta(p, q)$;

3. $Prob(x) := \sum_{p\in Q} f_p(n)$

Ausgabe: Die Wahrscheinlichkeit $Prob(x)$, dass $\mathcal{M}$ den String x generiert.

Dieser Algorithmus hat eine Laufzeit in $O(n\cdot|Q|)$ und läuft somit genauso schnell wie der Viterbi-Algorithmus.

Lehrstuhl für Informatik I
Prof. Juraj Hromkovič
Dr. Hans-J. Böckenhauer
Dirk Bongartz

9. Übung zur Vorlesung Algorithmische Grundlagen der Bioinformatik

Aachen, den 16. Juli 2003

Aufgabe 32

Wir wollen in dieser Aufgabe eine andere Charakterisierung additiver Bäume betrachten.
Sei A eine Menge von n Taxa, sei $\delta : A \times A \to \mathbb{Q}^{\geq 0}$ ein metrisches Distanzmaß auf A, so dass ein additiver Baum für A und δ existiert. Dann heißt δ *Baum-Metrik*.
Zeigen Sie, dass jede Baum-Metrik die folgende *Vier-Punkt-Bedingung* erfüllt:

Für alle $w, x, y, z \in A$ gilt

$$\delta(w,x) + \delta(y,z) \leq \max\{\delta(w,y) + \delta(x,z), \delta(w,z) + \delta(x,y)\}.$$

Bemerkung: Es gilt auch die Umkehrung, d. h. jede Metrik, die die Vier-Punkt-Bedingung erfüllt, ist eine Baum-Metrik. **10 Punkte**

Aufgabe 33

Geben Sie einen möglichst effizienten Algorithmus an, der das Parsimony-Problem mit Hilfe dynamischer Programmierung löst.
Hinweis: Behandeln Sie die einzelnen Positionen der Eingabestrings unabhängig, und berechnen Sie für jeden Teilbaum eine optimale Beschriftung bei vorgegebenem Symbol an der Wurzel des Teilbaums. **10 Punkte**

Aufgabe 34

Geben Sie einen möglichst effizienten Algorithmus an, der für zwei gegebene ungerichtete phylogenetische Bäume (gemäß Definition 11.12 der Vorlesung) entscheidet, ob diese isomorph sind oder nicht.
Hinweis: Für den Algorithmus ist es hilfreich, dass die Blätter der Bäume unterscheidbar sind.
10 Punkte

Aufgabe 35

Ein gerichteter Binärbaum ist ein Baum mit einer ausgezeichneten Wurzel, in dem alle Kanten von der Wurzel in Richtung der Blätter gerichtet sind, und in dem jeder innere Knoten einen Ausgangsgrad von 2 hat. Ein *gerichteter phylogenetischer Baum* für eine Menge A von Taxa ist ein gerichteter Binärbaum, dessen Blätter (bijektiv) mit den Taxa aus A beschriftet sind.
Zeigen Sie, dass für alle $n \geq 2$ die Anzahl der nicht isomorphen gerichteten phylogenetischen Bäume für n Taxa

$$\prod_{i=2}^{n}(2i-3) = \frac{(2n-3)!}{2^{n-2} \cdot (n-2)!}$$

beträgt. **10 Punkte**

Abgabe: bis Mittwoch, den 23. Juli 2003, um 14.00 Uhr im Sammelkasten am Lehrstuhl.

Lehrstuhl für Informatik I
Prof. Juraj Hromkovič
Dr. Hans-J. Böckenhauer
Dirk Bongartz

Musterlösung zur 9. Übung zur Vorlesung Algorithmische Grundlagen der Bioinformatik

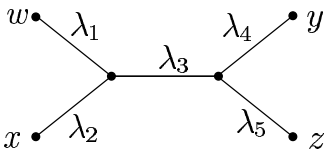
Aachen, den 25. Juni 2003

Lösung zu Aufgabe 32

Wir zeigen im Folgenden, dass jede Baum-Metrik die folgende Vier-Punkt-Bedingung erfüllt:

$$\text{Für alle } w,x,y,z \in A \text{ gilt: } \delta(w,x)+\delta(y,z) \leq \max\{\delta(w,y)+\delta(x,z), \delta(w,z)+\delta(x,y)\}.$$

Sei δ eine Baum-Metrik auf einer Menge A von Taxa. Dann existiert ein additiver Baum für A und δ. Seien $w,x,y,z \in A$ beliebig und paarweise verschieden. Dann ergibt sich o. B. d. A. die in der folgenden Abbildung dargestellte Situation, die λ_i geben dabei die Längen der einzelnen Teilpfade an (falls die Pfade von w nach y und von x nach z kantendisjunkt sind, gilt einfach $\lambda_3 = 0$).



Es ist offensichtlich, dass

$$\delta(w,x) + \delta(y,z) = (\lambda_1 + \lambda_2) + (\lambda_4 + \lambda_5)$$

und

$$\begin{aligned}\delta(w,y) + \delta(x,z) &= (\lambda_1 + \lambda_3 + \lambda_4) + (\lambda_2 + \lambda_3 + \lambda_5) \\ &= (\lambda_1 + \lambda_3 + \lambda_5) + (\lambda_2 + \lambda_3 + \lambda_4) \\ &= \delta(w,z) + \delta(x,y)\end{aligned}$$

Es gilt also

$$\delta(w,x) + \delta(y,z) \leq \delta(w,y) + \delta(x,z) = \delta(w,z) + \delta(x,y),$$

und somit auch

$$\delta(w,y) + \delta(x,z) \leq \max\{\delta(w,x) + \delta(y,z), \delta(w,z) + \delta(x,y)\}.$$

Damit ist die Vier-Punkt-Bedingung für w,x,y,z erfüllt. Falls die Taxa nicht paarweise verschieden sind, ergibt sich eine ähnliche Situation, in diesem Fall sind nur einige der Pfadlängen λ_i gleich.

Lösung zu Aufgabe 33

Um das Parsimony-Problem mit dynamischer Programmierung zu lösen, betrachten wir zunächst das Problem der optimalen Beschriftung mit nur einem Symbol für jeden Knoten. Das Gesamtproblem kann man dann dadurch lösen, dass man dieses kleinere Problem für jede Position der Eingabestrings löst und die verschiedenen Beschriftungen jedes Knotens entsprechend konkateniert.

Zunächst konstruiert man für das neue Problem wie im Fitch-Algorithmus einen Baum T' mit Wurzel aus dem Eingabebaum für das Parsimony-Problem, und konstruiert daraus (bei Stringlänge k) k einzelne Bäume T_i, $1 \leq i \leq k$, die sich von T' nur dadurch unterscheiden, dass ihre Blätter mit jeweils einem einzelnen Symbol beschriftet sind, das dem Symbol an Position i des entsprechenden Eingabestrings entspricht.

In jedem dieser Bäume beschriften wir die inneren Knoten derart, dass die Anzahl der Kanten, deren Endpunkte unterschiedlich beschriftet sind, minimiert wird.
Dieses Problem lösen wir für einen Baum $T = T_i$ für ein $i \in \{1, \ldots, k\}$ mit Hilfe dynamischer Programmierung. Hierfür verwenden wir die folgenden Notationen:

- T_v sei der Teilbaum von T mit Wurzel v für jeden Knoten v,
- $Cost(v)$ seien die Kosten der optimalen Lösung des Problems, beschränkt nur auf den Teilbaum T_v,
- $Cost(v, x)$ seien die Kosten der besten Beschriftung von T_v, bei der v mit x beschriftet ist,
- v_L [v_R] sei der linke [rechte] Sohn von v.
- $S(v)$ sei die endgültige Beschriftung des Knotens v,

Damit ergibt sich der folgende Algorithmus:

Eingabe: Ein Baum T mit Wurzel r, der n Knoten enthält und dessen Blätter mit einzelnen Symbolen aus Σ beschriftet sind.

1. Initialisierung:
   for all Blätter $v \in T$ **do**
   $Cost(v) := 0$
   for all $x \in \Sigma$ **do**
   if $x =$ Beschriftung von v **then**
   $Cost(v, x) := 0$
   else
   $Cost(v, x) := \infty$
   for all innere Knoten $k \in T$ **do**
   $Cost(k) := \infty$

2. Rekursion:
   for all innere Knoten (in Bottom-up-Reihenfolge) **do**
   for all $x \in \Sigma$ **do**
   $Cost(v, x) := \min\{Cost(v_L) + 1, Cost(v_L, x)\} + \min\{Cost(v_R) + 1, Cost(v_R, x)\}$;
   $Cost(v) := \min\{Cost(v, x) \mid x \in \Sigma\}$

3. Traceback:
   $S(r) := \text{argmin}_x\, Cost(r, x)$
   for all Knoten $v \in T$ (in Top-down-Reihenfolge) **do**
   if $Cost(v_L) = Cost(v_L, S(v))$ **then**
   $S(v_L) := S(v)$
   else
   $S(v_L) := \text{argmin}_x\, Cost(v_L, x)$
   if $Cost(v_R) = Cost(v_R, S(v))$ **then**
   $S(v_R) := S(v)$
   else
   $S(v_R) := \text{argmin}_x\, Cost(v_R, x)$

Ausgabe: Der durch S vollständig beschriftete Baum T.

Die Laufzeit des Algorithmus liegt in $O(n \cdot |\Sigma|)$. Da wir den Algorithmus zur Lösung des Parsimony-Problems k-mal ausführen müssen, liegt die Gesamtlaufzeit der Lösung in $O(k \cdot n \cdot |\Sigma|)$.

Lösung zu Aufgabe 34

Wir bezeichnen zwei phylogenetische (verwurzelte) Bäume T_1 mit Knotenmenge V_1 und T_2 mit Knotenmenge V_2 genau dann als isomorph, wenn es eine bijektive Abbildung $\varphi : V_1 \to V_2$ gibt, so dass für *jedes* Paar $u \in V_1, v := \varphi(u) \in V_2$ die Objekte, die in Blättern unterhalb von u enthalten sind, genau den Objekten in den Blättern unterhalb von v entsprechen.
Die grundlegende Idee hinter dem im Folgenden vorgestellten Algorithmus ist, die beiden Bäume „bottom-up", d. h. von den Blättern aufwärts zu vergleichen und dabei besuchte Knoten zu entfernen. Sobald wir festgestellt haben, dass zwei Paare von Geschwisterknoten (ein Paar in T_1 und das andere in T_2) isomorph sind, können wir diese aus dem Baum herausnehmen und ordnen dem Vaterknoten ein neues „künstliches" Objekt zu. Nachdem wir alle Blätter entfernt haben, werden deren Eltern selbst zu Blättern und wir wiederholen die Prozedur solange, bis wir alle Knoten betrachtet haben oder vorher die Nichtisomorphie der beiden Bäume gezeigt haben. Der Algorithmus beruht auf den folgenden Datenstrukturen und Funktionen:

- einer Menge S, die die aktuellen Blätter des Restbaums von T_1 enthält, deren Geschwister auch Blätter sind,
- eine Funktion $object(u)$, die das mit Knoten u assozierte Objekt zurückgibt,
- eine Funktion $sib(u)$, die den Geschwisterknoten (engl. *sibling*) des Knotens u liefert,
- eine Funktion $vater(u)$, die den Vater von Knoten u zurückgibt, sowie
- zwei Arrays $L_1[i]$ und $L_2[i]$ über Objekten, welche den Blattknoten im Baum T_1 respektive T_2 liefern, der das Objekt i enthält.

Der vollständige Algorithmus ist dann der folgende:

Eingabe: Zwei phylogenetische Bäume T_1 und T_2

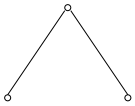
1. Initialisierung:
   $k := n$ {k zählt die Anzahl der Objekte plus „neuer" Objekte}
   $S := \emptyset$
   for all Blätter in T_1 **do**
   $L_1[object(u)] := u$
   if $sib(u)$ ist ein Blatt **then**
   $S := S \cup \{u\}$
   for all Blätter in T_2 **do**
   $L_2[object(v)] := v$
2. **while** $S \neq \emptyset$ **do**
   Entnehme einen beliebigen Knoten u aus S
   $u' := L_2[object(u)]$
   $y := sib(u); y' := sib(u')$
   if y' ist kein Blatt **or** $object(y') \neq object(y)$ **then**
   return `FALSE`
   $p_1 := vater(u); p_2 := vater(u')$
   Entferne Knoten u, y aus T_1 und u', y' aus T_2
   Markiere p_1 und p_2 als Blätter
   $k := k + 1$
   $L_1[k] := p_1; L_2[k] := p_2$ {p_1 und p_2 enthalten nun das „neue Objekt" k}
   if $sib(p_1)$ ist ein Blatt **then**
   $S := S \cup \{p_1\}$
   return `TRUE`

Die Laufzeit des Algorithmus ist offensichtlich linear in der Anzahl der Knoten, da wir lediglich konstant viele Male Bäume mit $O(n)$ Knoten traversieren und für jeden zu bearbeitenden Knoten konstante Zeit aufwenden.

Lösung zu Aufgabe 35

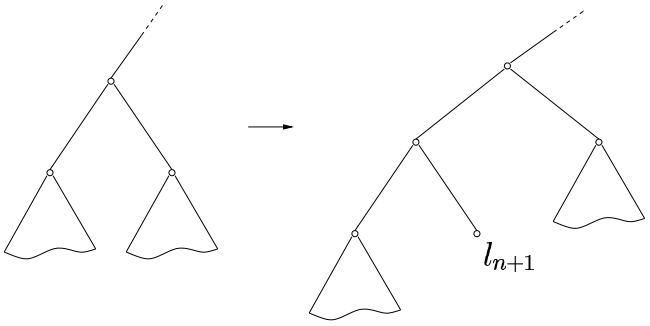
Wir zeigen die Aussage über die Anzahl der nicht isomorphen gerichteten phylogenetischen Bäume für n Taxa per Induktion über die Anzahl der Taxa, d.h. der Blätter des phylogenetischen Baumes.

Betrachten wir zunächst den Fall $n = 2$. In diesem Fall existiert genau ein gerichteter phylogenetischer Baum mit zwei Blättern, wie in der folgenden Abbildung dargestellt.



Bezeichne $\mathcal{T}_n$ die Menge der gerichteten phylogenetischen Bäume mit n Taxa, und es gelte $t(n) := |\mathcal{T}_n|$.

Man kann sich nun leicht überlegen, dass jeder gerichtete Binärbaum mit n Blättern genau $n-1$ innere Knoten haben muss. Genauso anschaulich ist klar, dass die Anzahl der Kanten eines jeden solchen Binärbäume um genau eins geringer ist als die Anzahl der Knoten, da bis auf die Wurzel in jeden Knoten genau eine Kante hineinführt. Verbindet man diese beiden Beobachtungen, so erhält man, dass jeder Binärbaum mit n Blättern $n+(n-1)-1 = 2n-2$ viele Kanten besitzt. Ein Baum aus $\mathcal{T}_{n+1}$ kann aus einem Baum aus $\mathcal{T}_n$ dadurch konstruiert werden, dass man das neu hinzuzufügende $(n+1)$-te Blatt an einen neuen inneren Knoten anhängt, welcher in der Mitte einer Kante des Baumes aus $\mathcal{T}_n$ eingefügt wird. Diese Einfüge-Operation ist in der folgenden Abbildung anschaulich dargestellt.



Da $t(n)$ Bäume in $\mathcal{T}_n$ enthalten sind, und jeder Baum aus $\mathcal{T}_n$ genau $2n-2$ Kanten besitzt, in die ein neuer inneren Knoten eingefügt werden könnte, gibt es hierfür genau $t(n) \cdot (2n-2)$ Möglichkeiten. Man kann das $(n+1)$-te Blatt jedoch auch an einen neuen Wurzelknoten anhängen, so dass der andere Sohn der neuen Wurzel die Wurzel eines Baumes in $\mathcal{T}_n$ ist. Offensichtlich führt dies zu $t(n)$ weiteren Bäumen und damit zu insgesamt $t(n) \cdot (2n-1)$ Bäumen, so dass wir die folgende Rekurrenzgleichung für die Anzahlen der Bäume erhalten:

$$t(n+1) = \begin{cases} 1 & \text{falls } n = 1, \\ t(n)(2n-1) & \text{falls } n > 1, \end{cases}$$

Man beachte, dass aufgrund der Blattbeschriftungen alle diese so konstruierten Bäume nicht-isomorph zueinander sind. Durch Auflösen der Rekurrenz erhält man

$$t(n) = 1 \cdot 3 \cdot 5 \cdots (2n-3).$$

Eine nützliche Hilfsformel ist nun

$$2 \cdot 4 \cdot 6 \cdots (2n-4) = 2^{n-2}[1 \cdot 2 \cdot 3 \cdots (n-2)],$$

und damit gilt für die Anzahl der gerichteten phylogenetischen Bäume mit n Knoten

$$t(n) = \frac{(2n-3)!}{2^{n-2}(n-2)!}.$$