INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN
LEHRSTUHL FÜR EFFIZIENTE ALGORITHMEN

Skriptum

zur Vorlesung

Algorithmische Bioinformatik I/II

gehalten im Wintersemester 2001/2002

und im Sommersemester 2002 von

Volker Heun

Erstellt unter Mithilfe von:

Peter Lücke – Hamed Behrouzi – Michael Engelhardt

Sabine Spreer – Hanjo Täubig

Jens Ernst – Moritz Maaß

14. Mai 2003

Version 0.96

Vorwort

Dieses Skript entstand parallel zu den Vorlesungen *Algorithmische Bioinformatik I* und *Algorithmische Bioinformatik II*, die im Wintersemester 2001/2002 sowie im Sommersemester 2002 für Studenten der Bioinformatik und Informatik sowie anderer Fachrichtungen an der Technischen Universität München im Rahmen des von der Ludwig-Maximilians-Universität und der Technischen Universität gemeinsam veranstalteten Studiengangs Bioinformatik gehalten wurde. Einige Teile des Skripts basieren auf der bereits im Sommersemester 2000 an der Technischen Universität München gehaltenen Vorlesung *Algorithmen der Bioinformatik* für Studierende der Informatik.

Das Skript selbst umfasst im Wesentlichen die grundlegenden Themen, die man im Bereich Algorithmische Bioinformatik einmal gehört haben sollte. Die vorliegende Version bedarf allerdings noch einer Ergänzung weiterer wichtiger Themen, die leider nicht in den Vorlesungen behandelt werden konnten.

An dieser Stelle möchte ich insbesondere Hamed Behrouzi, Michael Engelhardt und Peter Lücke danken, die an der Erstellung des ersten Teils dieses Skriptes (Kapitel 2 mit 5) maßgeblich beteiligt waren. Bei Sabine Spreer möchte ich mich für die Unterstützung bei Teilen des siebten Kapitels bedanken. Bei meinen Übungsleitern Jens Ernst und Moritz Maaß für deren Unterstützung der Durchführung des Übungsbetriebs, aus der einige Lösungen von Übungsaufgaben in dieses Text eingeflossen sind. Bei Hanjo Täubig möchte ich mich für die Mithilfe zur Fehlerfindung bedanken, insbesondere bei den biologischen Grundlagen.

Falls sich dennoch weitere (Tipp)Fehler unserer Aufmerksamkeit entzogen haben sollten, so bin ich für jeden Hinweis darauf (an heun@in.tum.de) dankbar.

München, im September 2002 Volker Heun

Inhaltsverzeichnis

Fragment Assembly

5.1 Sequenzierung ganzer Genome

In diesem Kapitel wollen wir uns mit algorithmischen Problemen beschäftigen, die bei der Sequenzierung ganzer Genome (bzw. einzelner Chromosome) auftreten. Im ersten Kapitel haben wir bereits biotechnologische Verfahren hierzu kennen gelernt. Nun geht es um die informatischen Methoden, um aus kurzen sequenzierten Fragmenten die Sequenz eines langen DNS-Stückes zu ermitteln.

5.1.1 Shotgun-Sequencing

Trotz des rasanten technologischen Fortschritts ist es nicht möglich, lange DNS-Sequenzen im Ganzen biotechnologisch zu sequenzieren. Selbst mit Hilfe großer Sequenzierautomaten lassen sich nur Sequenzen der Länge von etwa 500 Basenpaaren sequenzieren. Wie lässt sich dann allerdings beispielsweise das ganze menschliche Genom mit etwa drei Milliarden Basenpaaren sequenzieren?

Eine Möglichkeit ist das so genannte *Shotgun-Sequencing*. Hierbei werden lange Sequenzen in viele kurze Stücke aufgebrochen. Dabei werden die Sequenzen in mehrere Klassen aufgeteilt, so dass eine Bruchstelle in einer Klasse mitten in den Fragmenten der anderen Sequenzen einer anderen Klasse liegt (siehe auch Abbildung 5.1).

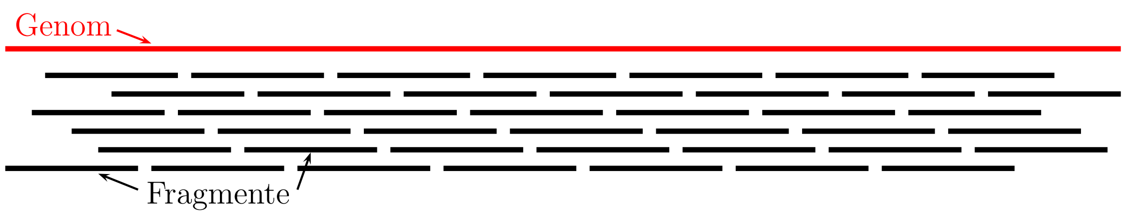

Abbildung 5.1: Skizze: Shotgun-Sequencing

Die kurzen Sequenzen können jetzt direkt automatisch sequenziert werden. Es bleibt nur das Problem, aus der Kenntnis der kurzen Sequenzen wieder die lange DNS-Sequenz zu rekonstruieren. Dabei hilft, dass einzelnen Positionen (oder vielmehr kurze DNS-Stücke) von mehreren verschiedenen Fragmenten, die an unterschiedlichen Positionen beginnen, überdeckt werden. Man muss also nur noch die Fragment wie in einem Puzzle-Spiel so anordnen, dass überlappende Bereiche möglichst gleich sind.

5.1.2 Sequence Assembly

Damit ergibt sich für das Shotgun-Sequencing die folgende prinzipielle Vorgehensweise:

Overlap-Detection Zuerst bestimmen wir für jedes Paar von zwei Fragmenten, wie gut diese beiden überlappen, d.h. für eine gegebene Menge $S = \{s_1, \ldots, s_k\}$ von k Fragmenten bestimmen wir für alle $i, j \in [1 : k]$ die beste Überlappung $d(s_i, s_j)$ zwischen dem Ende von s_i und dem Anfang von s_j (siehe Abbildung 5.2).

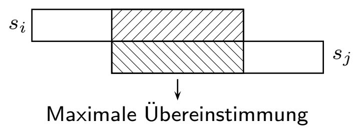

Abbildung 5.2: Skizze: Overlap-Detection

Fragment Layout Dann müssen die Fragmente so angeordnet werden, dass die Überlappungen möglichst gleich sind. Mit diesem Problem werden wir und in diesem Kapitel hauptsächlich beschäftigen.

Konsensus-String für gefundenes Layout Zum Schluss interpretieren wir das Fragment Layout wie ein mehrfaches Sequenzen Alignment und bestimmen den zugehörigen Konsensus-String, denn wir dann als die ermittelte Sequenz für die zu sequenzierende Sequenz betrachten.

In den folgenden Abschnitten gehen wir auf die einzelnen Schritte genauer ein. Im Folgenden werden wir mit $S = \{s_1, \ldots, s_k\}$ die Menge der Fragmente bezeichnen. Dabei werden in der Praxis die einzelnen Fragmente ungefähr die Länge 500 haben. Wir wollen ganz allgemein mit $n = \lfloor \frac{1}{k} \sum_{i=1}^{k} |S_i| \rfloor$ die mittlere Länge der Fragmente bezeichnen. Wir wollen annehmen, dass für alle Sequenzen in etwa gleich lang sind, also $|s_i| = \Theta(n)$ für alle $i \in [1 : k]$. In der Praxis gilt hierbei, dass kn in etwa 5 bis 10 Mal so groß ist wie die Länge der zu sequenzierenden Sequenz, da wir bei der Generierung der Fragmente darauf achten werden, dass jede Position der zu sequenzierenden Sequenz von etwa 5 bis 10 verschiedenen Fragmenten überdeckt wird.

5.2 Overlap-Detection und Fragment-Layout

Zuerst wollen wir uns mit der Overlap-Detection beschäftigen. Wir wollen hier zwei verschiedene Alternativen unterscheiden, je nachdem, ob wir Fehler zulassen wollen oder nicht.

5.2.1 Overlap-Detection mit Fehlern

Beim Sequenzieren der Fragmente treten in der Regel Fehler auf, so dass man damit rechnen muss. Ziel wird es daher sein, einen Suffix von s_i zu bestimmen, der ziemlich gut mit einem Präfix von s_j übereinstimmt. Da wir hierbei (Sequenzier-)Fehler zulassen, entspricht dies im Wesentlichen einem semi-globalen Alignment. Hierbei

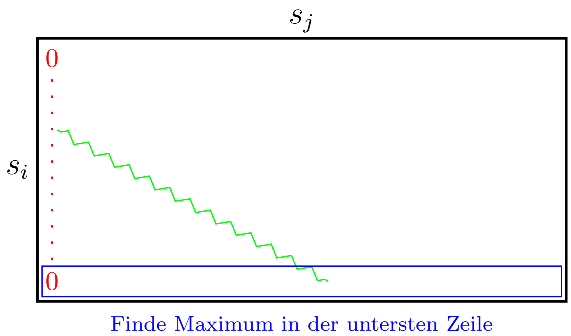

Abbildung 5.3: Semi-globale Alignments mit Ähnlichkeitsmaßen

ist zu beachten, dass Einfügungen zu Beginn von s_i und Löschungen am Ende von s_j nicht bestraft werden. Für den Zeitbedarf gilt (wie wir ja schon gesehen haben) $O(n^2)$ pro Paar. Da wir $k^2 - k$ verschiedene Paare betrachten müssen, ergibt dies insgesamt eine Laufzeit von: $O(k^2n^2)$.

Theorem 5.1 *Sei $S = \{s_1, \ldots, s_k\}$ eine Menge von Sequenzen mit $|s_i| = \Theta(n)$ für alle $i \in [1 : k]$. Die längsten Überlappung für jedes Paar (s_i, s_j) für alle $i, j \in [1 : k]$ mit einem vorgegebenen beschränkten Fehler kann in Zeit $O((kn)^2)$ berechnet werden.*

5.2.2 Overlap-Detection ohne Fehler

Wir wollen jetzt noch eine effizientere Variante vorstellen, wenn wir keine Fehler zulassen. Dazu verwenden wir wieder einmal Suffix-Bäume.

5.2.2.1 Definition von $L(v)$

Wir konstruieren zuerst einen verallgemeinerter Suffix-Baum für $S = \{s_1, \ldots, s_k\}$. Wie wir schon gesehen haben ist der Platzbedarf $\Theta(kn)$ und die Laufzeit der Konstruktion $O(kn)$. Für jeden Knoten v des Suffix-Baumes generieren wir eine Liste

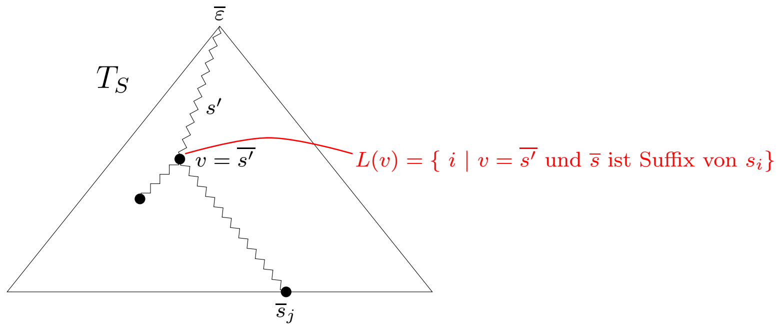

Abbildung 5.4: Skizze: Verallgemeinerter Suffix-Baum für $S = \{s_1, \ldots, s_k\}$

$L(v)$, die wie folgt definiert ist:

$$L(v) := \left\{ i \in [1:k] \ : \ \exists j \in [1:|s_i|] : v = \overline{s_{i,j} \cdots s_{i,|s_i|}} \right\}.$$

In der Liste $L(v)$ befinden sich also alle Indizes i, so dass ein Suffix von s_i im Knoten v endet.

Betrachten wir also jetzt einen Knoten v im verallgemeinerten Suffix-Baum mit seiner Liste $L(v)$, wie in Abbildung 5.4 illustriert. Sei dabei s' die Zeichenfolge, mit der der Knoten s' erreicht wird. Dann gilt offensichtlich:

- s' ist Suffix von s_i für alle $i \in L(v)$,
- s' ist Präfix von s_j für alle j, so dass $\overline{s}_j$ ein Blatt im vom v gewurzelten Teilbaum ist.

Der längste Suffix-Präfix-Match von s_i mit s_j ist damit durch den tiefsten Knoten v mit $i \in L(v)$ auf einem Pfad von $\overline{\varepsilon}$ zu $\overline{s}_j$ gegeben.

5.2.2.2 Erzeugung von $L(v)$

Überlegen wir uns jetzt, wie man diese Listen effizient erstellen kann. Für alle $s_i \in S$ tun wir das Folgende. Starte an $\overline{s}_i$ und folge den Suffix-Links bis zur Wurzel (Implizites Durchlaufen aller Suffixe von s). Für jeden besuchten Knoten v füge i in die Liste eine: $L(v) := L(v) \cup \{i\}$. Die Kosten hierfür entsprechen der Anzahl der Suffixe von s_i, dies sind $|s_i| + 1 = O(n)$ viele. Für alle $s \in S$ mit $|S| = k$ ist dann der Zeitbedarf insgesamt $O(kn)$.

5.2.2.3 Auffinden längster Suffix-Präfix-Matches

Wie finden wir jetzt mit Hilfe dieses verallgemeinerten Suffix-Baumes und den Listen längste Suffix-Präfix-Matches? Für jedes $i \in [1 : k]$ legen wir einen Keller $S[i]$ an. Wenn wir mit einer Tiefensuche den verallgemeinerten Suffix-Baum durchlaufen, soll folgendes gelten. Befinden wir uns am Knoten w des verallgemeinerten Suffix-Baumes, dann soll der Stack $S[i]$ alle Knoten v beinhalten, die zum einen Vorfahren von w sind und zum anderen soll in v ein Suffix von s_i enden.

Wenn wir jetzt eine Tiefensuche durch den verallgemeinerten Suffixbaum durchführen, werden wir für jeden neu aufgesuchten Knoten zuerst die Stack aktualisieren, d.h. wir füllen die Stacks geeignet auf. Wenn nach der Abarbeitung des Knotens wieder im Baum aufsteigen, entfernen wir die Elemente wieder, die wir beim ersten Besuch des Knotens auf die Stacks gelegt haben. Nach Definition einer Tiefensuche, müssen sich diese Knoten wieder oben auf den Stacks befinden.

```
SUFFIX-PREFIX-MATCHES (int[] S)
{
    tree T_S;   /* verallgemeinerter Suffixbaum T_S */
    stack_of_nodes S[k];   /* je einen für jedes s_i ∈ S */
    int level[V(T_S)];
    int Overlap[k, k];
    level[ε̄] = 0;
    DFS(T_S, ε̄);
}

DFS (tree T, node v)
{
    for all (i ∈ L(v)) do S[i].push(v);
    if (v = s̄_j)
        for all (i ∈ [1 : k]) do
            if (not S[i].isEmpty())
                Overlap(s_i, s_j) = level[S[i].top()]};
    for all (v, w) do
    {
        level[w] = level[v] + 1;
        DFS(w);
    }
    for all (i ∈ L(v)) do S[i].pop();
}
```

Abbildung 5.5: Algorithmus: modifizierte Depth-First-Search

Sobald wir ein Blatt gefunden haben, das ohne Beschränkung der Allgemeint zum String s_j gehört, holen wir für jedes $i \in [1 : k]$ das oberste Element w vom Stack (sofern eines existiert). Da dies das zuletzt auf den Stack gelegte war, war dieses eines, das den längsten Überlappung von s_i mit s_j ausgemacht hat. Wenn wir für v noch den Level mitberechnen, entspricht der Level einem längsten Overlap. Wie üblich ist der Level der Wurzel 0, und der Level eines Knoten ist um eines größer, als der Level seines Elters. Damit ergibt sich zur Lösung der in Abbildung 5.5 angegebene Algorithmus.

Kommen wir nun zur Bestimmung der Laufzeit. Für den reinen DFS-Anteil (der grüne Teil) der Prozedur benötigen wir Zeit $O(kn)$. Die Kosten für die Pushs und Pops auf die Stacks (der schwarze Teil) betragen:

$$\sum_{v \in V(T_S)} |L(v)| = \left| \left\{ t \in \Sigma^* \ : \ \exists s \in S : \exists j \in [1 : |s|] : t = s_j \cdots s_{|s|} \right\} \right| = O(kn),$$

da in der zweiten Menge alle Suffixe von Wörtern aus S auftauchen.

Die Aktualisierungen der Overlaps (der rote Teil) verursachen folgende Kosten. Da insgesamt nur k Knoten eine Sequenz aus S darstellen, wird die äußere if-Anweisung insgesamt nur $O(k)$ mal betreten. Darin wird innere for-Schleife jeweils k mal aufgerufen. Da die Aktualisierung in konstanter Zeit erledigt werden kann, ist der gesamte Zeitbedarf $O(k^2)$.

Theorem 5.2 *Sei $S = \{s_1, \ldots, s_k\}$ eine Menge von Sequenzen mit $|s_i| = \Theta(n)$ für alle $i \in [1 : k]$. Die längsten Überlappung für jedes Paar (s_i, s_j) für alle $i, j \in [1 : k]$ kann in Zeit $O(nk + k^2)$ berechnet werden.*

An dieser Stelle sei noch darauf hingewiesen, dass die Rechenzeit im vorherigen Theorem optimal ist.

5.2.3 Greedy-Ansatz für das Fragment-Layout

Das Fragment-Layout kann man beispielsweise mit Hilfe eines Greedy-Algorithmus aufbauen. Hierbei werden in das Fragment Layout die Overlaps in der Reihenfolge nach ihrem Score eingearbeitet, beginnend mit dem Overlap mit dem größten Score.

Sind beide Sequenzen noch nicht im Layout enthalten, so werden sie mit dem aktuell betrachteten Overlap in dieses neu aufgenommen. Ist eine der beiden Sequenzen bereits im Layout enthalten, so wird die andere Sequenz mit dem aktuell betrachteten Overlap in das Layout aufgenommen. Hierbei muss beachtet werden, wie sich die neue Seqeunz in das bereits konstruierte aufnehmen lässt. Kommt es hier zu großen Widersprüchen, so wird die Seqeunz mit dem betrachteten Overlap nicht aufgenommen. Sind bereits beide Sequenzen im Overlap enthalten und befinden sich in verschiedenen Zusammenhangskomponenten des Layouts, so wird versucht die beiden Komponenten mit dem aktuell betrachteten Overlap zusammenzufügen. Auch hier wird der betrachtete Overlap verworfen, wenn sich mit diesem Overlap die beiden bereits konstruierten Layout nur mit großen Problemen zusammenfügen lassen.

Bei dieser Vorgehensweise gibt es insbesondere Probleme bei so genannten Repeats. Repeats sind Teilsequenzen die mehrfach auftreten. Bei Prokaryonten ist dies eher selten, bei Eukaryonten treten jedoch sehr viele Repeats auf. In der Regel werden dann solche Repeats, die ja mehrfach in der Sequenz auftauchen, auf eine Stelle im Konsensus abgebildet. Ist die vorhergesagte Sequenz deutlich zu kurz, so deutet dies auf eine fehlerhafte Einordnung von Repeats hin.

5.3 Shortest Superstring Problem

In diesem Abschnitt wollen wir das so genannte *Shortest Superstring Problem* (*SSP*) und eine algorithmische Lösung hierfür vorstellen. Formal ist das Problem wie folgt definiert.

Geg.: $S = \{s_1, \ldots, s_k\}$
Ges.: $s^* \in \Sigma$, so dass s_i Teilwort von s^* und $|s^*|$ minimal ist

Dies ist eine Formalisierung des Fragment Assembly Problems. Allerdings gehen wir hierbei davon aus, dass die Sequenzierung fehlerfrei funktioniert hat. Ansonsten müssten die Fragmente nur sehr ähnlich zu Teilwörtern des Superstrings, aber nicht identisch sein. Ferner nehmen wir an, dass die gefunden Überlappungen auch wirklich echt sind und nicht zufällig sind. Zumindest bei langen Überlappungen kann man davon jedoch mit hoher Wahrscheinlichkeit ausgehen. Bei kurzen Überlappungen (etwa bei 5 Basenpaaren), kann dies jedoch auch rein zufällig sein. Wie wir später sehen werden, werden wir daher auch den längeren Überlappungen ein größeres Vertrauen schenken als den kürzeren.

Obwohl wir hier die Existenz von Fehlern negieren, ist das Problem und dessen Lösung nicht nur von theoretischem Interesse. Auch bei vorhandenen Fehlern wird die zugrunde liegende Lösungsstrategie von allgemeinem Interesse sein, da diese prinzipiell auch beim Vorhandensein von Fehlern angewendet werden kann.

Zuerst die schlechte Nachricht: Das Shortest Superstring Problem ist $\mathcal{NP}$-hart. Wir können also nicht hoffen, dass wir eine optimale Lösung in polynomieller Zeit finden können. Wie schon früher werden wir versuchen, eine möglichst gute Näherungslösung zu finden. Leider ist das SSP auch noch $\mathcal{APX}$-hart. Die Komplexitätsklasse $\mathcal{APX}$ umfasst alle Optimierungsprobleme, die man in polynomieller Zeit bis auf einen konstanten Faktor approximieren kann. Gehört nun ein Problem zu den schwierigsten Problemen der Klasse $\mathcal{APX}$ (ist also $\mathcal{APX}$-hart bezüglich einer geeigneten Reduktion, für Details verweisen wir auf Vorlesungen über Komplexitätstheorie), dann gibt es eine Zahl $\alpha > 1$, so dass einen Näherungslösung die eine Approximation bis auf einen Faktor kleiner als α liefert, nicht in polynomieller Zeit konstruiert werden kann (außer $\mathcal{P} = \mathcal{NP}$).

Im Folgenden wollen wir zeigen, dass mithilfe einer Greedy-Strategie eine Lösung gefunden werden kann, die höchstens viermal so lang wie eine optimale Lösung ist. Mithilfe derselben Idee und etwas mehr technischen Aufwand, lässt sich sogar eine 2,5-Approximation finden.

5.3.1 Ein Approximationsalgorithmus

Für die Lösung des SSP wollen wir in Zukunft ohne Beschränkung der Allgemeinheit annehmen, dass kein s_i Teilwort von s_j für $i \neq j$ sei (andernfalls ist s_i ja bereits in einem Superstring für $S \setminus \{s_i\}$ als Teilwort enthalten).

Definition 5.3 *Sei $s, t \in \Sigma^*$ und sei v das längste Wort aus Σ^*, so dass es $u, w \in \Sigma^+$ mit $s = uv$ und $t = vw$ gibt. Dann bezeichne*

- $o(s,t) = v$ *den* Overlap *von s und t,*
- $p(s,t) = u$ *das* Präfix *von s in t.*

Abbildung 5.6: Skizze: Overlap und Präfix von s und t

In der Abbildung 5.6 ist der Overlap $v := o(s,t)$ von s und t sowie das Präfix $u = p(s,t)$ von s in t noch einmal graphisch dargestellt. Beachte, dass der Overlap ein echtes Teilwort von s und t sein muss. Daher ist der Overlap von $s = aabab$ und $t = abab$ eben $o(s,t) = ab$ und nicht $abab$. Dies spielt hier keine allzu große Rolle, da wir zu Beginn dieses Abschnitts ohne Beschränkung der Allgemeinheit angenommen haben, dass kein Wort Teilwort eines anderen Wortes der gegebenen Menge ist. Dies ist jedoch wichtig, wenn wir den Overlap und den Präfix eines Wortes mit sich selbst berechnen wollen. Beispielsweise ist für $s = aaa$ der Overlap $o(s,s) = aa$ und somit $p(s,s) = a$ sowie für $s' = abbab$ ist der Overlap sogar das leere Wort: $o(s',s') = \varepsilon$. Wir wollen an dieser Stelle noch die folgende einfache, aber wichtige Beziehung festhalten.

Lemma 5.4 *Sei $s,t \in \Sigma^*$, dann gilt*

$$s = p(s,t) \cdot o(s,t).$$

In der Abbildung 5.7 ist ein Beispiel zur Illustration der obigen Definitionen anhand von drei Sequenzen angegeben.

Bsp:	s_1=ACACG	$o(s_1,s_1) = \varepsilon$	$p(s_1,s_1)$ =ACACG
	s_2=ACGTT	$o(s_1,s_2)$ =ACG	$p(s_1,s_1)$ =AC
	s_3=GTTA	$o(s_1,s_3)$ =G	$p(s_1,s_3)$ =ACAC
		$o(s_2,s_1) = \varepsilon$	$p(s_2,s_1)$ =ACGTT
		$o(s_2,s_2) = \varepsilon$	$p(s_2,s_2)$ =ACGTT
		$o(s_2,s_3)$ =GTT	$p(s_2,s_3)$ =AC
		$o(s_3,s_1)$ =A	$p(s_3,s_1)$ =GTT
		$o(s_3,s_2)$ =A	$p(s_3,s_2)$ =GTT
		$o(s_3,s_3) = \varepsilon$	$p(s_3,s_3)$ =GTTA

Abbildung 5.7: Beispiel: Overlaps und Präfixe

Lemma 5.5 *Sei $S = \{s_1, \ldots, s_k\}$, dann gilt für eine beliebige Permutation der Indizes $(i_1, \ldots, i_k) \in \mathcal{S}(k)$, dass*

$$p(s_{i_1}, s_{i_2}) \cdots p(s_{i_2}, s_{i_3}) \cdots p(s_{i_{k-1}}, s_{i_k}) \cdot s_{i_k}$$

ein Superstring von S ist.

Beweis: Wir führen den Beweis mittels Induktion über k.

Induktionsanfang ($k = 1$): Hierfür ist die Aussage trivial, da $p(s_1, s_1) \cdot s_1$ offensichtlich s_1 als Teilwort enthält.

Induktionsschritt ($k \to k+1$): Nach Induktionsvoraussetzung gilt

$$s' = \underbrace{p(s_{i_1}, s_{i_2}) \cdots p(s_{i_{k-1}}, s_{i_k})}_{s''} \cdot s_{i_k},$$

wobei s' ein Superstring für $\{s_{i_1}, \ldots, s_{i_k}\}$ ist.

Nach Lemma 5.4 ist $s_{i_k} = p(s_{i_k}, s_{i_{k+1}}) \cdot o(s_{i_k}, s_{i_{k+1}})$. Daher enthält $p(s_{i_k}, s_{i_{k+1}}) \cdot s_{i_k}$ sowohl s_{i_k} als auch $s_{i_{k+1}}$, da $o(s_{i_k}, s_{i_{k+1}})$ ein Präfix von $s_{i_{k+1}}$ ist. Dies ist in der Abbildung 5.8noch einmal graphisch dargestellt. Also ist

$$p(s_{i_1}, s_{i_2}) \cdots p(s_{i_{k-1}}, s_{i_k}) \cdot s_{i_k}$$

ein Superstring für die Zeichenreihen in $S = \{s_1, \ldots, s_{k+1}\}$. ■

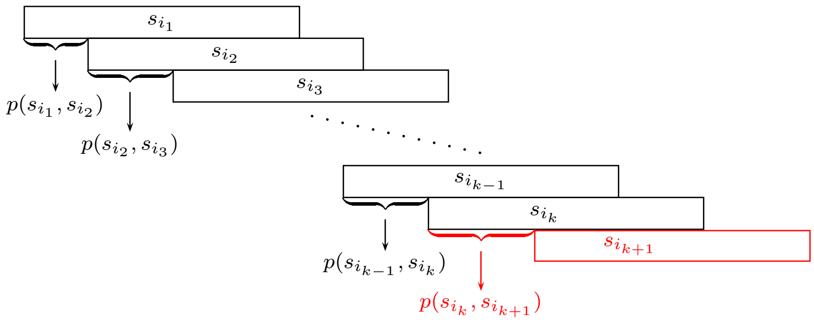

Abbildung 5.8: Skizze: Erweiterung des Superstrings

Korollar 5.6 *Sei $S = \{s_1, \ldots, s_k\}$, dann ist für eine beliebige Permutation der Indizes $(i_1, \ldots, i_k) \in S(k)$ die Zeichenfolge*

$$p(s_{i_1}, s_{i_2}) \cdots p(s_{i_{k-1}}, s_{i_k}) \cdot p(s_{i_k}, s_{i_1}) \cdot o(s_{i_k}, s_{i_1})$$

ein Superstring.

Beweis: Dies folgt aus dem vorhergehenden Lemma und dem Lemma 5.4, das besagt, dass $s_{i_k} = p(s_{i_k}, s_1) \cdot o(s_{i_k}, s_1)$. ■

Lemma 5.7 *Sei $S = \{s_1, \ldots, s_k\}$ und s^* der kürzeste Superstring für S. Dann gibt es eine Permutation der Indizes $(i_1, \ldots, i_k) \in S(k)$ mit*

$$s^* = p(s_{i_1}, s_{i_2}) \cdots p(s_{i_{k-1}}, s_{i_k}) \cdot s_{i_k}.$$

Beweis: Sei s^* ein (kürzester) Superstring für $S = \{s_1, \ldots, s_k\}$. Wir definieren a_i als die kleinste ganze Zahl, so dass $s^*_{a_i} \cdots s^*_{a_i+|s_i|-1} = s_i$ gilt. Umgangssprachlich ist

a_i die erste Position, an der s_i als Teilwort von s^* auftritt. Da s^* ein Superstring von $S = \{s_1, \ldots, s_k\}$ ist, sind alle a_i für $i \in [1 : k]$ wohldefiniert. Wir merken noch an, dass die a_i paarweise verschieden sind, da wir ja ohne Beschränkung der Allgemeinheit angenommen haben, dass in s kein Wort Teilwort eines anderen Wortes ist.

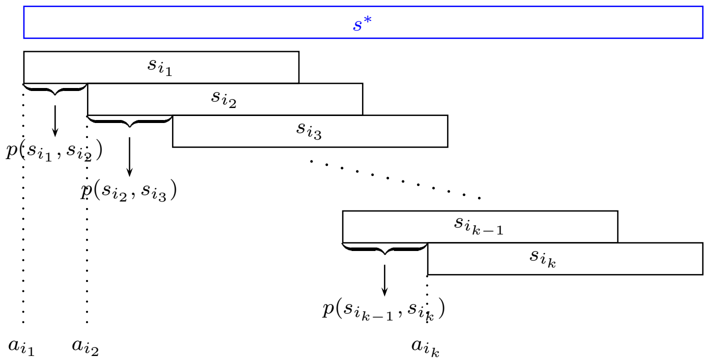

Abbildung 5.9: Skizze: Auffinden der s_i im Superstring

Sei nun $(i_1, \ldots, i_k) \in S(k)$ eine Permutation über $[1 : k]$, so dass

$$a_{i_1} < a_{i_2} < \cdots < a_{i_k}.$$

Dann ist $(i_1, \ldots, i_k)$ die gesuchte Permutation. Dies ist in Abbildung 5.9 noch einmal illustriert. Man beachte, dass $a_{i_1} = 1$ und $a_{i_k} + |s_{i_k}| - 1 = |s^*|$ gilt, da sonst s^* nicht der kürzeste Superstring von S wäre. ■

Korollar 5.8 *Sei $S = \{s_1, \ldots, s_k\}$ und s^* der kürzeste Superstring für S. Dann gibt es eine Permutation der Indizes $(i_1, \ldots, i_k) \in S(k)$ mit*

$$s^* = p(s_{i_1}, s_{i_2}) \cdots p(s_{i_{k-1}}, s_{i_k}) \cdot p(s_{i_k}, s_{i_1}) \cdot o(s_{i_k}, s_{i_1}).$$

Aus den beiden letzten Korollaren folgt, dass den Präfixen von je zwei Zeichenreihen ineinander eine besondere Bedeutung für eine kürzesten Superstring zukommt. Dies motiviert die folgenden Definition eines Präfix-Graphen.

Definition 5.9 *Der gewichtete gerichtete Graph $G_S = (V, E, \gamma)$ für eine Menge S von Sequenzen $S = \{s_1, \ldots, s_k\}$ heißt* Präfix-Graph *von S, wobei $V = S$, $E = V \times V = S \times S$ und das Gewicht für $(s, t) \in E$ durch $\gamma(s, t) = |p(s, t)|$ definiert ist.*

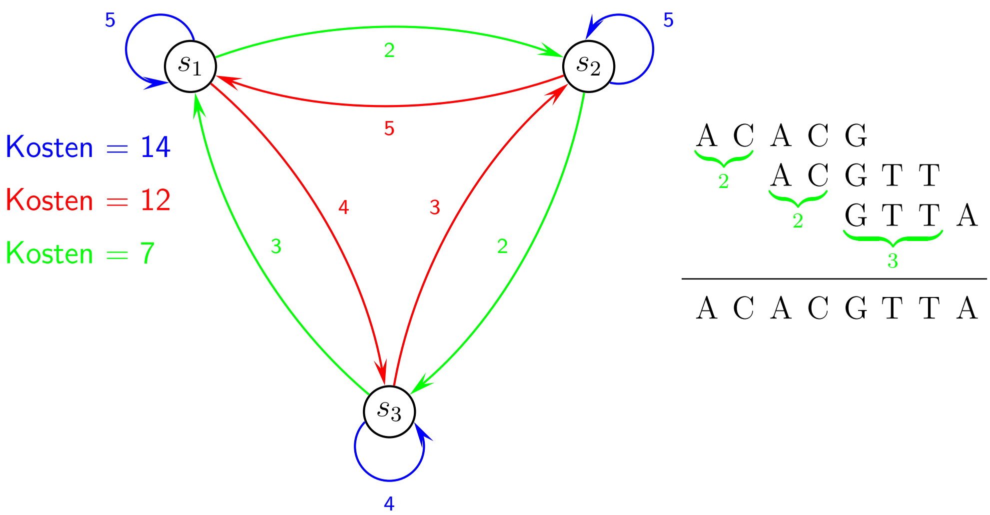

Abbildung 5.10: Beispiel: Zyklenüberdeckung für {ACACG,ACGTT,GTTA}

In Abbildung 5.10 ist der Präfix-Graphen samt einiger Zyklenüberdeckungen für das vorherige Beispiel dargestellt. Natürlich ist in diesem Beispiel auch (s_1, s_2) und s_3 eine Zyklenüberdeckung mit einem Gewicht von $2 + 3 + 4 = 9$.

5.3.2 Hamiltonsche Kreise und Zyklenüberdeckungen

Definition 5.10 *Sei $G = (V, E)$ ein Graph. Ein Pfad $(v_1, \ldots, v_k) \in V^k$ heißt* hamiltonsch, *wenn $(v_{i-1}, v_i) \in E$ für alle $i \in [2 : k]$ ist und $\{v_1, \ldots, v_k\} = V$ sowie $|V| = k$ gilt. $(v_1, \ldots, v_k) \in V^k$ ist ein* hamiltonscher Kreis, *wenn $(v_1, \ldots, v_k)$ ein hamiltonscher Pfad ist und wenn $(v_k, v_1) \in E$ gilt. Ein Graph heißt* hamiltonsch, *wenn er einen hamiltonschen Kreis besitzt.*

Damit haben wir eine wichtige Beziehung gefunden: Superstrings von S und hamiltonsche Kreise im Präfixgraphen von S korrespondieren zueinander. Das Gewicht eines hamiltonschen Kreises im Präfix-Graphen von S entspricht fast der Länge des zugehörigen Superstrings, nämlich bis auf $|o(s_j, s_{(j \bmod k)+1})|$, je nachdem, an welcher Stelle j man den hamiltonschen Kreis aufschneidet.

Im Folgenden werden wir also statt kürzester Superstrings für S kürzeste hamiltonsche Kreis im entsprechenden Präfix-Graphen suchen. Dabei werden wir von der Hoffnung geleitet, dass die Länge des gewichteten hamiltonschen Kreises im Wesentliche der Länge des kürzesten Superstrings entspricht und die Größe $|o(s_j, s_{(j \bmod k)+1})|$ ohne spürbaren Qualitätsverlust vernachlässigt werden kann.

Definition 5.11 *Sei $S = \{s_1, \ldots, s_k\}$ und G_S der zugehörige Präfix-Graph. $C(G_S)$ bezeichnet den kürzesten (bzgl. des Gewichtes) Hamiltonschen Kreis in G_S:*

$$C(G_S) := min\left\{\sum_{j=1}^{k} |p(s_{i_j}, s_{i_{j+1}})| \;:\; (s_{i_1}, \ldots, s_{i_k}) \in \mathcal{H}(G_S)\right\},$$

wobei $\mathcal{H}(G)$ die Menge aller hamiltonscher Kreise in einem Graphen G bezeichnet.

Definition 5.12 *Sei $S = \{s_1, \ldots, s_k\}$ und sei S^* die Menge aller Superstrings von S. Dann bezeichnet*

$$SSP(S) := min\left\{|s| \;:\; s \in S^*\right\}$$

die Länge eines kürzesten Superstrings für S.

Das folgende Korollar fasst die eben gefundene Beziehung noch einmal zusammen.

Korollar 5.13 *Sei $S = \{s_1, \ldots, s_k\}$ und G_S der zugehörige Präfix-Graph, dann gilt $C(G_S) \leq SSP(S)$.*

Leider ist die Berechnung von hamiltonschen Kreisen mit minimalem Gewicht ebenfalls ein algorithmisch schwer lösbares Problem. In der Literatur ist es als *Traveling Salesperson Problem(TSP)* bekannt und ist $\mathcal{NP}$-hart. Daher gibt es auch wieder keine optimale Lösung, die sich in polynomieller Zeit berechnen lässt (außer, wenn $\mathcal{P} = \mathcal{NP}$ gilt). Daher werden wir die Problemstellung etwas relaxieren und zeigen, dass wir dafür eine optimale Lösung in polynomieller Zeit berechnen können. Leider wird die optimale Lösung des relaxierten Problems nur eine Näherungslösung für das ursprüngliche Problem liefern

Für die weiteren Untersuchungen wiederholen wir noch ein paar elementare graphentheoretische Bezeichnungen. Sei im Folgenden $G = (V, E)$ ein ungerichteter Graph. Für $v \in V$ bezeichnen wir mit $N(v) = \{w : \{v, w\} \in E\}$ die *Nachbarschaft* des Knotens v. Mit dem *Grad* $d(v) := |N(v)|$ des Knotens v bezeichen wir die Anzahl seiner Nachbarn. Einen Knoten mit Grad 0 nennen wir einen *isolierter Knoten*. Mit

$$\begin{aligned} \Delta(G) &:= \max\left\{d(v) \;:\; v \in V\right\} \qquad \text{bzw.} \\ \delta(G) &:= \min\left\{d(v) \;:\; v \in V\right\} \end{aligned}$$

bezeichnen wir den *Maximal-* bzw. *Minimalgrad* eines Knotens in G.

Im Falle gerichteter Graphen gibt es folgenden Ergänzungen und Modifikationen. Sei also im Folgenden $G = (V, E)$ ein gerichteter Graph. Die Menge der Nachbarn eines Knotens v bezeichnen wir weiterhin mit $N(v)$. Wir unterteilen die Nachbarschaft in die Menge der direkten Nachfolger und der direkten Vorgänger, die wir mit $N^+(v)$ und $N^-(v)$ bezeichnen wollen:

$$\begin{aligned} N^+(v) &= \{w \in V : (v, w) \in E\}, \\ N^-(v) &= \{w \in V : (w, v) \in E\}, \\ N(v) &= N^+(v) \cup N^-(v). \end{aligned}$$

Der *Eingangsgrad* bzw. *Ausgangsgrad* eines Knotens $v \in V(G)$ ist die Anzahl seiner direkten Vorgänger bzw. Nachfolger und wird mit $d^- = |N^-(v)|$ bzw. $d^+ = |N^+(v)|$ bezeichnet. Der *Grad* eines Knotens $v \in V(G)$ ist definiert als $d(v) := d^-(v) + d^+(v)$ und es gilt somit $d \geq |N(v)|$. Mit

$$\begin{aligned} \Delta(G) &:= \max\{d(v) : v \in V\} \qquad \text{bzw.} \\ \delta(G) &:= \min\{d(v) : v \in V\} \end{aligned}$$

bezeichnen wir den *Maximal-* bzw. *Minimalgrad* eines Knotens in G. Mit

$$\begin{aligned} \Delta^-(G) &:= \max\left\{d^-(v) : v \in V\right\} \qquad \text{bzw.} \\ \delta^-(G) &:= \min\left\{d^-(v) : v \in V\right\} \end{aligned}$$

bezeichnen wir den *maximalen* bzw. *minimalen Eingangsgrad* eines Knotens in G. Analog bezeichen wir mit

$$\begin{aligned} \Delta^+(G) &:= \max\left\{d^+(v) : v \in V\right\} \qquad \text{bzw.} \\ \delta^+(G) &:= \min\left\{d^+(v) : v \in V\right\} \end{aligned}$$

den *maximalen* bzw. *minimalen Ausgangsgrad* eines Knotens in G.

Definition 5.14 *Sei $G = (V, E)$ ein gerichteter Graph. Eine* Zyklenüberdeckung *(engl.* cycle cover*) von G ist ein Teilgraph $C = (V', E')$ mit den folgenden Eigenschaften:*

- *$V' = V$,*
- *$E' \subseteq E$,*
- *$\Delta^+(G) = \Delta^-(G) = \delta^+(G) = \delta^-(G) = 1$.*

Mit $\mathcal{C}(G)$ bezeichnen wir die Menge aller Zyklenüberdeckungen von G.

Ist C eine Zyklenüberdeckung von G, dann bezeichne C_i mit $C = \bigcup_i C_i$ die einzelnen Zusammenhangskomponenten von C. Dabei ist dann jede Komponente C_i ein gerichteter Kreis.

Definition 5.15 *Sei* $S = \{s_1, \ldots, s_k\}$ *und* G_S *der zugehörige Präfix-Graph. Dann bezeichnet* $CS(G_S)$ *das Gewicht einer minimalen Zyklenüberdeckung:*

$$CS(G_S) := min\left\{\sum_{i=1}^{r} \gamma(C_i) \; : \; C = \bigcup_i C_i \wedge C \in \mathcal{C}(G_S)\right\}.$$

Notation 5.16 *Sei* $S = \{s_1, \ldots, s_k\}$ *und* G_s *der zugehörige Präfix-Graph. Weiter sei* $C = \bigcup_{i=1}^{r} C_i$ *eine Zyklenüberdeckung für* G_S*. Dann bezeichne*

$$\begin{aligned} \ell(C_i) &:= \ell_i := \max\{|s_j| \; : \; s_j \in V(C_i)\}, \\ w(C_i) &:= w_i := \gamma(C_i) = \sum_{c \in E(C_i)} \gamma(c) = \sum_{(u,v) \in E(C_i)} |p(u,v)|. \end{aligned}$$

Lemma 5.17 *Ist* s' *ein Superstring von* $S = \{s_1, \ldots, s_k\}$*, der aus einer Zyklenüberdeckung* $C = \bigcup_{i=1}^{r} C_i$ *konstruiert wurde, dann gilt:*

$$\sum_{i=1}^{r} w_i \leq |s'| \leq \sum_{i=1}^{r} (w_i + \ell_i).$$

5.3.3 Berechnung einer optimalen Zyklenüberdeckung

In diese Abschnitt wollen wir nun zeigen, dass sich eine optimale Zyklenüberdeckung effizient berechnen lässt. Dazu benötigen wir der einfacheren Beschreibung wegen noch eine Definition.

Definition 5.18 *Der gewichtete gerichtete Graph* $B_S = (V, E, \gamma)$ *für eine Menge von Sequenzen* $S = \{s_1, \ldots, s_k\}$ *heißt* Overlap-Graph *von* S*, wobei:*

- $V = S \cup S'$ *wobei* $S' = \{s' \; : \; s \in S\}$ *mit* $S \cap S' = \emptyset$,
- $E = \{\{s, s'\} \; : \; s \in S \wedge s' \in S'\}$,
- $\gamma(s, t') = |o(s,t)| = |s| - |p(s,t)|$ *für* $s, t \in S$.

In Abbildung 5.11 ist der Overlap-Graph B_S für unser bereits bekannten Beispielsequenzen angegeben.

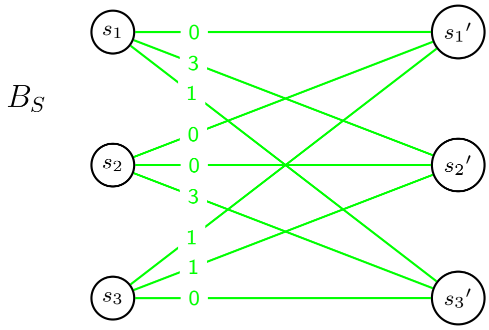

Abbildung 5.11: Beispiel: Overlap-Graph für {ACACG,ACGTT,GTTA}

Es drängt sich nun die Idee auf, dass minimale Zyklenüberdeckungen in G_S gerade gewichtsmaximalen Matchings in B_S entsprechen. Ob dies nun tatsächlich der Fall ist, soll im Folgenden untersucht werden.

Definition 5.19 *Sei $G = (V, E)$ ein ungerichteter Graph. Eine Kantenmenge $M \subseteq E$ heißt* Matching, *wenn für den Graphen $G(M) = (V, M)$ gilt, dass $\Delta(G(M)) = 1$ und $\delta(G(M) \geq 0$. Eine Kantenmenge M heißt* perfektes Matching, *wenn gilt, dass $\Delta(G(M)) = 1$ und $\delta(G(M)) = 1$.*

Man beachte, dass nur Graphen mit einer geraden Anzahl von Knoten ein perfektes Matching besitzen können. Im Graphen in der Abbildung 5.12 entsprechen die rot hervorgehobenen Kanten einem perfekten Matching M des Graphen G_S.

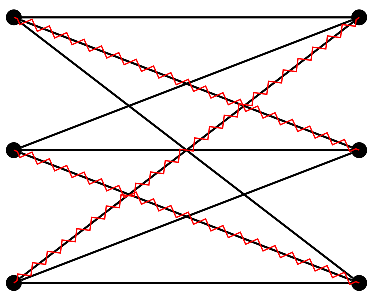

Abbildung 5.12: Beispiel: Matching in G_S

Aus einer Zyklenüberdeckung im Präfix-Graphen können wir sehr einfach ein perfektes Matching in einem Overlap-Graphen konstruieren. Für jede Kante (s, t) in der Zyklenüberdeckung im Präfix-Graphen nehmen wir $\{s, t'\}$ in das Matching des Overlap-Graphen auf. Umgekehrt können wir eine Zyklenüberdeckung im Präfix-Graphen aus einen Matching des Overlap-Graphen konstruieren, indem wir für jede

Matching-Kante $\{s, t'\}$ die gerichtete Kante (s, t) in die Zyklenüberdeckung aufnehmen. Man überlegt sich leicht, dass man aus der Zyklenüberdeckung im Präfix-Graphen ein perfektes Matching im Overlap-Graphen erhält und umgekehrt.

In der folgenden Abbildung 5.13 wird nochmals der Zusammenhang zwischen einem minimalen CC in G_S und einem gewichtsmaximalen Matching in B_S anhand des bereits bekannten Beispiels illustriert. Hier ist der Übersichtlichkeit halber ein Zyklus und das korrespondierende perfekte Matching auf den entsprechende Knoten besonders hervorgehoben.

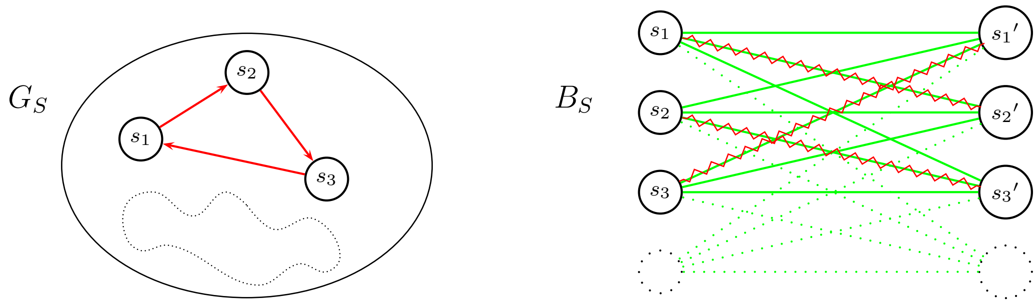

Abbildung 5.13: Skizze: Cycle Cover in G_S entspricht perfektem Matching in B_S

Sei $C = \bigcup_{i=1}^{r} C_i$ eine Zyklenüberdeckung von G_S. Es gilt dann:

$$\gamma(C) = \sum_{(u,v)\in E(C)} |p(u,v)|.$$

Für ein perfektes Matching M in B_s erhalten wir entsprechend:

$$\begin{aligned}
\gamma(M) &= \sum_{(u,v)\in M} |o(u,v)| \\
&= \sum_{(u,v)\in M} (\underbrace{|u| - |p(u,v)|}_{|o(u,v)|}) \\
&= \underbrace{\sum_{u\in S} |u|}_{=:N} - \sum_{(u,v)\in M} |p(u,v)|.
\end{aligned}$$

Damit erhalten wir, dass für ein zu einer Zyklenüberdeckung C in G_s korrespondierendes perfektes Matching M in B_S gilt:

$$\begin{aligned}
M &= \{(u,v) \,:\, (u,v) \in E(C)\}, \\
\gamma(M) &= N - \gamma(C).
\end{aligned}$$

Umgekehrt erhalten wir, dass für eine zu einem perfekten Matching M in B_s korrespondierende Zyklenüberdeckung C in G_S gilt:

$$\begin{aligned} C &= \{(u,v) \,:\, (u,v') \in M\}, \\ \gamma(C) &= N - \gamma(M). \end{aligned}$$

Hierbei ist $N = \sum_{s \in S} |s|$ eine nur von der Menge $S = \{s_1, \ldots, s_k\}$ abhängige Konstante. Somit können wir nicht aus der Zyklenüberdeckungen im Präfix-Graphen sehr einfach ein perfektes Matching konstruieren und umgekehrt, sondern auch die gewichteten Werte der auseinander konstruierten Teilgraphen lassen sich sehr leicht berechnen. Fassen wir das im folgenden Satz noch einmal zusammen.

Theorem 5.20 *Sei $S = \{s_1, \ldots, s_k\}$ eine Menge von Sequenzen über Σ mit $N = \sum_{s \in S} |s|$ und G_S bzw. B_S der zugehörige Präfix- bzw. Overlap-Graph. Zu jeder minimalen Zyklenüberdeckung C in G_S existiert ein gewichtsmaximales Matching M in B_S mit $\gamma(M) = N - \gamma(C)$ und zu jedem gewichtsmaximalen Matching M in B_S existiert eine minimale Zyklenüberdeckung C in G_S mit $\gamma(C) = N - \gamma(M)$.*

5.3.4 Berechnung gewichtsmaximaler Matchings

Wenn wir nun ein gewichtsmaximales Matching in B_s gefunden haben, haben wir also sofort eine Zyklenüberdeckung von G_S mit minimalem Gewicht. Zum Auffinden eines gewichtsmaximalen perfekten Matchings in B_S werden wir wieder einmal einen Greedy-Ansatz verwenden. Wir sortieren zunächst die Kanten absteigend nach ihrem Gewicht. Dann testen wir in dieser Reihenfolge jede Kante, ob wir diese zu unserem bereits konstruierten Matching hinzunehmen dürfen, um ein Matching mit einer größeren Kardinalität zu erhalten. Dieser Ansatz ist noch einmal im Pseudo-Code in Abbildung 5.14 angegeben.

Zunächst einmal halten wir fest, dass wir immer ein perfektes Matching erhalten. Dies folgt unmittelbar aus dem Heiratssatz (oder auch Satz von Hall). Für die Details verweisen wir auf die entsprechenden Vorlesungen oder die einschlägige Literatur. Wir werden später noch zeigen, dass wir wirklich ein gewichtsmaximales Matching erhalten, da der Graph B_S spezielle Eigenschaften aufweist, die mit Hilfe eines Greedy-Ansatzes eine optimale Lösung zulassen. Wir merken an dieser Stelle noch kurz an, dass es für bipartite Graphen einen Algorithmus zum Auffinden gewichtsmaximaler Matchings gibt, der eine Laufzeit von $O(k^3)$ besitzt. Auf die Details dieses Algorithmus sei an dieser Stelle auf die einschlägige Literatur verwiesen.

```
W_Max_Matching (graph (V, E, γ))
{
    set M = ∅;
    Sortiere E nach den Gewichten γ                          → O(m log m)
    E = {e_1, ..., e_m} mit γ(e_1) ≥ ··· ≥ γ(e_m)
    for (i = 1; i ≤ m; i++)                                  → O(m)
        if (e_i ist zu keiner anderen Kante aus M inzident)
            M = M ∪ {e_i};
    return M;
}
```

Abbildung 5.14: Algorithmus: Greedy-Methode für ein gewichtsmaximales Matching

Nun wollen wir uns um die Laufzeit unseres Greedy-Algorithmus kümmern. Wir werden zeigen, dass er ein besseres Laufzeitverhalten als $O(k^3)$ besitzt. Für das Sortieren der k^2 Kantengewichte benötigen wir eine Laufzeit von $O(k^2 \log(k))$. Das Abtesten jeder einzelnen Kante, ob sie zum Matching hinzugefügt werden darf, kann in konstanter Zeit realisiert werden. Somit ist die gesamte Laufzeit $O(k^2 \log(k))$.

Wenn man annehmen kann, dass die Kantengewichte nur aus einem kleinen Intervall möglicher Werte vorkommen, so kann man die Sortierphase mit Hilfe eines Bucket-Sorts noch auf $O(k^2)$ beschleunigen. Auch hier verweisen wir für die Details auf die einschlägige Literatur.

Lemma 5.21 *Der Greedy-Algorithmus liefert für einen gewichteten vollständigen bipartiten Graphen auf $2k$ Knoten in Zeit $O(k^2 \log(k))$ ein gewichtsmaximales Matching.*

Jetzt wollen wir uns nur noch darum kümmern, dass der Greedy-Algorithmus wirklich ein gewichtsmaximales Matching findet. Dazu benötigen wir die so genannte Monge-Ungleichung.

Definition 5.22 *Sei $G = (A, B, E, \gamma)$ ein gewichteter vollständiger bipartiter Graph mit $E = \{\{a, b\} : a \in A \wedge b \in B\}$ und $\gamma : E \to \mathbb{R}$. Der Graph G erfüllt die* Monge-Ungleichung *oder* Monge-Bedingung, *wenn für beliebige vier Knoten $s, p \in A$ und $t, q \in B$, mit $\gamma(s, t) \geq \max\{\gamma(s, q), \gamma(p, t), \gamma(p, q)\}$ gilt, dass*

$$\gamma((s, t)) + \gamma((p, q)) \geq \gamma((s, q)) + \gamma((p, t)).$$

Die Monge-Bedingung ist in der folgenden Abbildung 5.15 illustriert. Anschaulich besagt diese, dass auf Knoten das perfekte Matching mit der gewichtsmaximalen

Kante ein Gewicht besitzt, das mindestens so groß ist wie das andere mögliche perfekte Matching auf diesen vier Knoten.

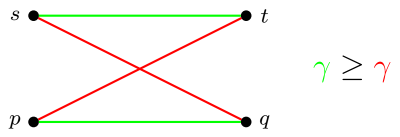

Abbildung 5.15: Skizze: Monge-Bedingung

Wir werden zuerst zeigen, dass unser Overlap-Graph B_S die Monge-Bedingung erfüllt und anschließend, dass der Greedy-Algorithmus zur Bestimmung gewichtsmaximaler Matchings auf gewichteten vollständigen bipartiten Graphen mit der Monge-Bedingung eine optimale Lösung liefert.

Lemma 5.23 *Sei $S = \{s_1, \ldots, s_k\}$ und B_S der zugehörige Overlap-Graph. Dann erfüllt B_S die Monge-Ungleichung.*

Beweis: Seien $s, p \in A$ und t, q beliebige vier Knoten des Overlap-Graphen B_S, so dass die für die Monge-Ungleichung die Kante (s, t) maximales Gewicht besitzt. Insbesondere gelte $\gamma(s,t) \geq \max\{\gamma(s,q), \gamma(p,t), \gamma(p,q)\}$. Betrachten wir die vier zugehörigen Kanten und ihre entsprechenden Zeichenreihen aus S. Der Einfachheit wegen identifizieren wir die Knoten des Overlap-Graphen mit den entsprechenden Zeichenreihen aus S. Da die Kantengewichte gleich den Overlaps der Zeichenreihen sind, ergibt sich das folgende in Abbildung 5.16 illustrierte Bild.

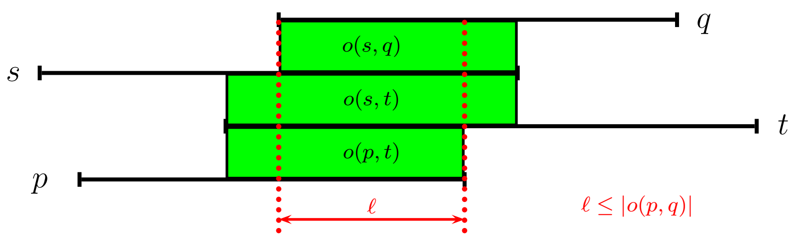

Abbildung 5.16: Skizze: Overlap-Graph erfüllt Monge-Bedingung

Da s und t nach Voraussetzung den längsten Overlaps (das maximale Gewicht) besitzen, kann der Overlap von s und q sowie von p und t nicht länger sein (grüne Bereiche im Bild). Betrachten man nun den roten Bereich der Länge ℓ, so stellt man fest, dass hier sowohl s und q sowie s und t als auch p und t übereinstimmen.

Daher muss in diesem Bereich auch p und q übereinstimmen und wir haben eine untere Schranke für $|o(p, q)|$ gefunden. Man beachte, dass der rote Bereich ein echtes Teilwort ist (wie in der Definition des Overlaps gefordert). Daher können wir sofort folgern, dass gilt:

$$|o(s, t)| + |o(p, q)| \geq |o(s, q)| + |o(p, t)|.$$

Damit gilt dann auch, dass $\gamma(s, t) + \gamma(p, q) \geq \gamma(s, q) + \gamma(p, t)$ und das Lemma ist bewiesen. ■

Theorem 5.24 *Sei $G = (A, B, E, \gamma)$ ein gewichteter vollständiger bipartiter Graph mit $E = \{\{a, b\} : a \in A \wedge b \in B\}$ und $\gamma : E \to \mathbb{R}_+$. Der Greedy-Algorithmus für gewichtsmaximale Matchings in G liefert eine optimale Lösung.*

Beweis: Wir führen den Beweis durch Widerspruch. Sei M das Matching, das vom Greedy-Algorithmus konstruiert wurde und sei M^* ein optimales Matching in B_S, d.h wir nehmen an, dass $\gamma(M^*) > \gamma(M)$. Wir wählen unter allen möglichen Gegenbeispielen ein „kleinstes“ (so genannter kleinster Verbrecher), d.h. wir wählen die Ausgabe M eines Ablaufs des Greedy-Algorithmus, so dass $|M^* \triangle M|$ minimal ist. Hierbei bezeichnet $A \triangle B := (A \setminus B) \cup (B \setminus A)$ die symmetrische Differenz von A und B.

Da $\gamma(M^*) < \gamma(M)$ muss $M^* \neq M$ sein. Da alle Kanten nichtnegativ sind, können wir ohne Beschränkung der Allgemeinheit annehmen, dass ein gewichtsmaximales Matching auch perfekt sein muss.

Wir wählen jetzt eine gewichtsmaximale Kante aus, die im Matching des Greedy-Algorithmus enthalten ist, die aber nicht im optimalen Matching ist, d.h. wir wählen $(s, t) \in M \setminus M^*$, so dass $\gamma(s, t)$ maximal unter diesen ist.

Da (wie oben bereits angemerkt) das gewichtsmaximale Matching perfekt sein muss, muss M^* zwei Kanten beinhalten, die die Knoten s und t überdecken. Also seien p und q so gewählt, dass $\{s, q\} \in M^*$ und $\{p, t\} \in M^*$ gilt. Zusätzlich betrachten wir noch die Kante $\{p, q\}$, die nicht im optimalen Matching M^* enthalten ist. Die Kante $\{p, q\}$ kann, muss aber nicht im Matching des Greedy-Algorithmus enthalten sein. Diese vier Knoten und Kanten sind in der Abbildung 5.17 noch einmal illustriert.

Da $\{s, t\}$ eine schwerste Kante aus $M \setminus M^*$ ist, muss

$$\gamma(s, t) \geq \gamma(s, q) \quad \wedge \quad \gamma(s, t) \geq (p, t)$$

gelten, da der Greedy-Algorithmus ansonsten $\{s, q\}$ oder $\{p, t\}$ anstatt $\{s, t\}$ gewählt hätte. Man sollte hier noch anmerken, dass zu diesem Zeitpunkt der Greedy-Algorithmus keine Kante ins Matching aufgenommen hat, die p oder q überdeckt. Eine

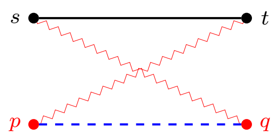

Abbildung 5.17: Skizze: Widerspruchsbeweis zur Optimalität

solche Kante wäre ebenfalls in $M \setminus M^*$ und da die Kante (s,t) mindestens eine schwerste ist, wird diese vom Greedy-Algorithmus zuerst gewählt.

Da für den Overlap-Graphen B_S die Monge-Ungleichung erfüllt ist, gilt

$$\gamma((s,t)) + \gamma((p,q)) \ \geq \ \gamma((s,q)) + \gamma((p,t)).$$

Wir betrachten nun folgende Menge $M' := M^* \setminus \{(s,q),(p,t)\} \cup \{(s,t),(p,q)\}$. Offensichtlich ist M' ein perfektes Matching von B_S. Aus der Monge-Ungleichung folgt, dass $\gamma(M') \geq \gamma(M^*)$. Da M^* ein gewichtsmaximales Matching war, gilt also $\gamma(M') = \gamma(M^*)$. Offensichtlich gilt aber auch

$$|M' \triangle M| \ < \ |M^* \triangle M|.$$

Dies ist der gewünschte Widerspruch, da wir ja mit M einen kleinsten Verbrecher als Gegenbeispiel gewählt haben und jetzt angeblich M' ein noch kleinere Verbrecher wäre. ■

5.3.5 Greedy-Algorithmus liefert eine 4-Approximation

Bis jetzt haben wir jetzt gezeigt, dass wir eine Näherungslösung für das SSP effizient gefunden haben. Wir müssen jetzt noch die Güte der Näherung abschätzen. Hierfür müssen wir erst noch ein noch ein paar grundlegende Definitionen und elementare Beziehungen für periodische Zeichenreihen zur Verfügung stellen.

Definition 5.25 *Ein Wort $s \in \Sigma^*$ hat eine* Periode p, *wenn $p < |s|$ ist und es gilt:*

$$\forall i \in [1 : |s| - p] : \ s_i = s_{i+p}.$$

Man sagt dann auch, s besitzt die Periode p.

Beispielsweise besitzt $w = aaaaaa$ die Periode 3, aber auch jede andere Periode aus $[1 : |w| - 1]$. Das Wort $w' = ababc$ besitzt hingegen gar keine Periode.

Zunächst werden wir zeigen, dass zwei verschiedene Perioden für dasselbe Wort gewisse Konsequenzen für die kleinste Periode dieses Wortes hat. Im Folgenden bezeichnet $\mathrm{ggT}(a, b)$ für zwei natürliche Zahlen $a, b \in \mathbb{N}$ den größten gemeinsamen Teiler: $\mathrm{ggT}(a, b) = \max\{k \in \mathbb{N} : (k \mid a) \wedge (k \mid b)\}$, wobei $k \mid a$ gilt, wenn es ein $n \in \mathbb{N}$ gibt, so dass $n \cdot k = a$.

Lemma 5.26 (GGT-Lemma für Zeichenreihen) *Sei $s \in \Sigma^*$ ein Wort mit Periode p und mit Periode q, wobei $p > q$ und $p + q \leq |s|$. Dann hat s auch eine Periode von $\mathrm{ggT}(p, q)$.*

Beweis: Wir zeigen zunächst, dass das Wort s auch die Periode $p - q$ besitzt. Dazu unterscheiden wir zwei Fälle, je nachdem, wie sich i zu q verhält.

Fall 1 ($i \leq q$): Da $i \leq q$ ist ist $i + p \leq p + q \leq |s|$. Weiter besitzt s die Periode p und es gilt $s_i = s_{i+p}$. Da $p > q$ ist, gilt $p - q > 0$ und somit ist $i + p - q > i$. Weiter besitzt s auch die Periode q und es $s_{i+p} = s_{i+p-q}$. Insgesamt ist also $s_i = s_{i+(p-q)}$ für $i \leq q$. Dies ist in der folgenden Abbildung illustriert.

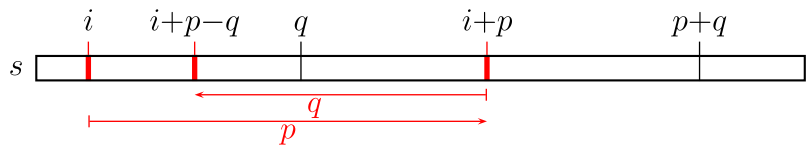

Abbildung 5.18: Skizze: 1. Fall $i \leq q$

Fall 2 ($i > q$): Da $i > q$ ist ist $i - q \geq 1$. Weiter besitzt s die Periode q und es gilt $s_i = s_{i-q}$. Da $p > q$ ist, gilt $p - q > 0$ und somit ist $i + p - q > i$. Weiter besitzt s auch die Periode p und es $s_{i-q} = s_{i+p-q}$. Insgesamt ist also $s_i = s_{i+(p-q)}$ für $i \leq q$. Dies ist in der folgenden Abbildung illustriert.

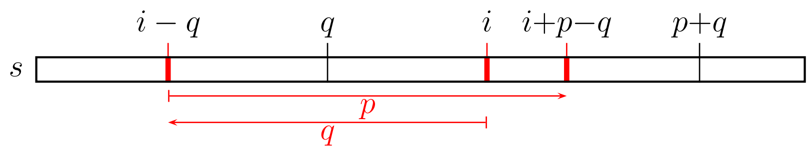

Abbildung 5.19: Skizze: 2.Fall $i > q$

Damit gilt für alle $i \in [1 : |s| - (p - q)]$, dass $s_i = s_{i+(p-q)}$. Somit besitzt s auch die Periode $p - q$.

Erinnern wir uns an den Euklidischen Algorithmus. Dieser ist für $p > q$ durch $\mathrm{ggT}(p, q) = \mathrm{ggT}(q, p - q)$ rekursiv definiert; dabei ist die Abbruchbedingung durch $\mathrm{ggT}(p, p) = p$ gegeben. Da die Perioden von s dieselbe Rekursionsgleichung erfüllen, muss also auch $\mathrm{ggT}(p, q)$ eine Periode von s sein. ■

Mithilfe dieses Lemmas können wir jetzt eine nahe liegende, jedoch für die Approximierbarkeit wichtige Eigenschaft von einer optimalen Zyklenüberdeckung beweisen. Diese besagt umgangssprachlich, dass zwei Wörter, die in verschiedenen Kreisen der Zyklenüberdeckung vorkommen, keine allzu große Überlappung besitzen können.

Lemma 5.27 (Overlap-Lemma) *Sei $S = \{s_1, \ldots, s_k\}$ und G_S der zugehörige Präfix-Graph. Sei $C = \bigcup_{i=1}^{r} C_i$ eine gewichtsminimale Zyklenüberdeckung für G_S. Sei $t_i \in V(C_i)$ und $t_j \in V(C_j)$ mit $i \neq j$. Dann gilt:*

$$|o(t_i, t_j)| \leq w_i + w_j = \gamma(C_i) + \gamma(C_j).$$

Beweis: Wir führen den Beweis durch Widerspruch und nehmen hierzu an, dass $|o(t_i, t_j)| > w_i + w_j$. Wir beobachten dann das Folgende:

- t_i hat Periode w_i und damit hat auch $o(t_i, t_j)$ die Periode w_i;
- t_j hat Periode w_j und damit hat auch $o(t_i, t_j)$ die Periode w_j;
- Es gilt $|t_i| > |o(t_i, t_j)| > w_i + w_j \geq w_i$;
- Es gilt $|t_j| > |o(t_i, t_j)| > w_i + w_j \geq w_j$.

Wir betrachten jetzt alle Knoten im Kreis C_i; genauer betrachten wir alle Wörter, deren korrespondierende Knoten sich im Kreis C_i befinden, siehe dazu die folgende Skizze in Abbildung 5.20. Hier sind die Wörter entsprechend der Reihenfolge des Auftretens in C_i beginnend mit t_i angeordnet. Der Betrag der Verschiebung der Wörter entspricht gerade der Länge des Präfixes im vorausgehenden zum aktuell betrachteten Wort in C_i. Man beachte, dass auch die Wortenden monoton aufsteigend sind. Andernfalls wäre ein Wort Teilwort eines anderen Wortes, was wir zu Beginn dieses Abschnitts ausgeschlossen haben.

Sind alle Wörter des Kreises einmal aufgetragen, so wird das letzte Wort am Ende noch einmal wiederholt. Da die Wörter in den überlappenden Bereichen übereinstimmen (Definition des Overlaps), kann man aus dem Kreis den zugehörigen Superstring für die Wörter aus C_i ableiten und dieser muss eine Periode von w_i besitzen.

Wir unterscheiden jetzt zwei Fälle, je nachdem, ob $w_i = w_j$ ist oder nicht.

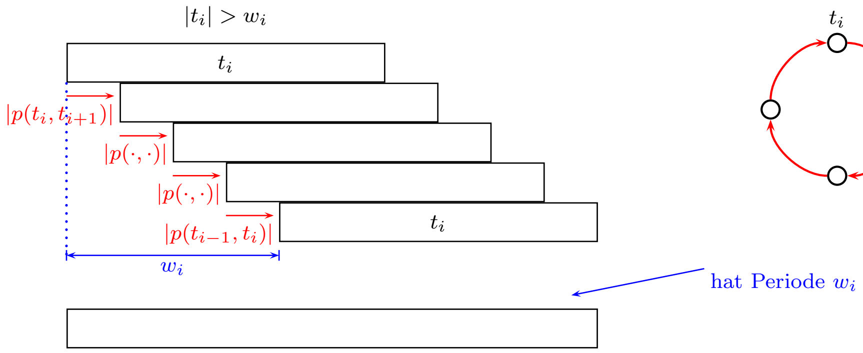

Abbildung 5.20: Skizze: Wörter eines Zyklus

Fall 1 ($w_i = w_j$): Da nun beide Zyklen die gleiche Periode besitzen können wir diese in einen neuen Zyklus zusammenfassen. Da der Overlap von t_i und t_j nach Widerspruchsannahme größer als $2w_i$ ist und beide Wörter die Periode w_i besitzen, muss das Wort t_j in den Zyklus von t_i einzupassen sind. Siehe dazu auch Abbildung 5.21. Man kann also die beiden Zyklen zu einem verschmelzen, so dass das

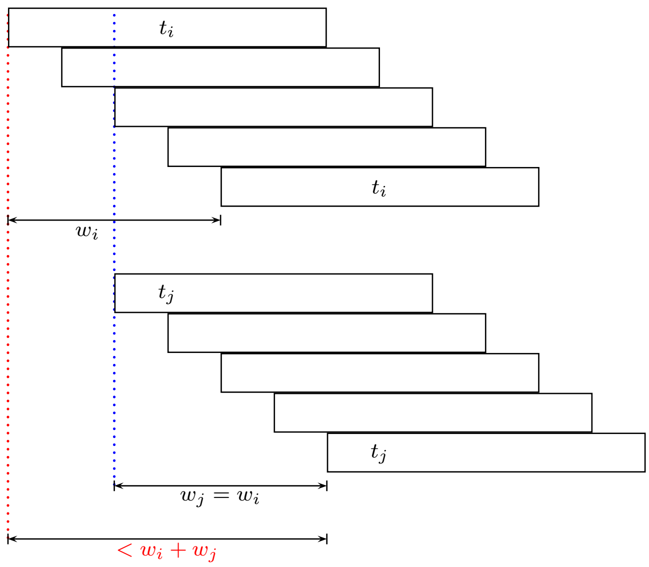

Abbildung 5.21: Skizze: 2 Zyklen mit demselben Gewicht

Gewicht des Zyklus kleiner als $2w_i = w_i + w_j$ wäre. Dies ist aber ein Widerspruch zur Optimalität der Zyklenüberdeckung.

Fall 2 ($w_i > w_j$): Jetzt ist sicherlich $w_i \neq w_j$ und außerdem ist $|o(t_i, t_j)| \geq w_i + w_j$. Also folgt mit dem GGT-Theorem, dass $o(t_i, t_j)$ eine Periode von $g := \text{ggT}(w_i, w_j)$ besitzt. Da t_i auch die Periode w_i besitzt und g ein Teiler von w_i ist, muss das ganze

Wort t_i die Periode g besitzen. Dies ist Abbildung 5.22 veranschaulicht. Eine analoge Überlegung gilt natürlich auch für t_j, so dass also auch t_j eine Periode von g besitzt.

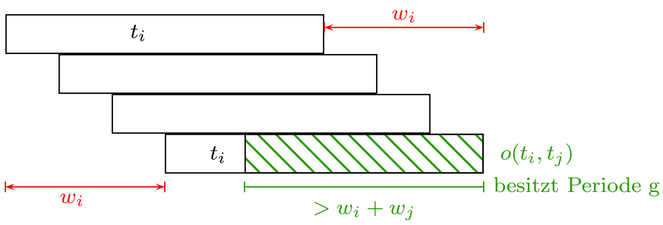

Abbildung 5.22: Skizze: Übertragung der Periode g von $o(t_i, t_j)$ auf t_i

Wir werden auch jetzt wieder zeigen, dass sich die beiden Zyklen die t_i bzw. t_j beinhalten zu einem neuen Zyklus verschmelzen lassen, dessen Gewicht geringer als die Summe der beiden Gewichte der ursprünglichen Zyklen ist. Dazu betrachten wir die Illustration in Abbildung 5.23. Da sowohl t_i als auch t_j einen Periode von g

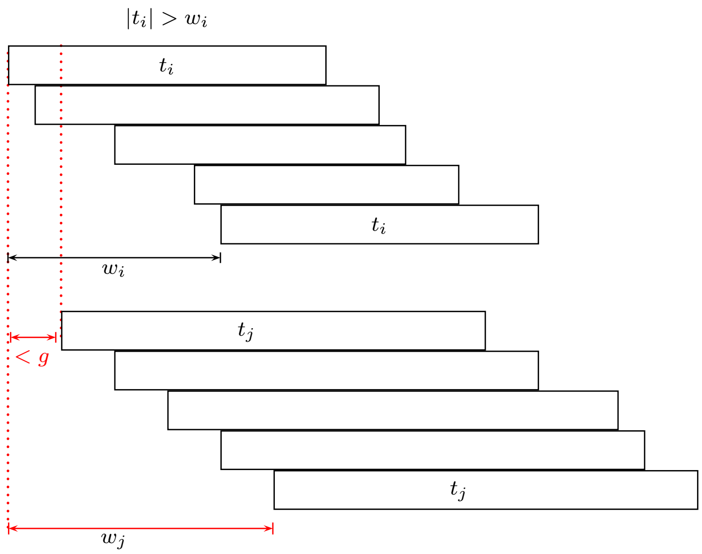

Abbildung 5.23: Skizze: Verschmelzung zweier Zyklen mit $w_i \neq w_j$

besitzen und die Zeichen innerhalb der Periode (die ja auch innerhalb des Overlaps liegt) gleich sind, lässt sich der Zyklus, der t_j enthält, in den Zyklus, der t_i enthält, integrieren. Somit hat der neue Zyklus ein Gewicht von $g + w_j < w_i + w_j$, was offensichtlich ein Widerspruch zur Optimalität der Zyklenüberdeckung ist. Somit ist das Overlap-Lemma bewiesen. ∎

Mit Hilfe des eben bewiesenen Overlap-Lemmas können wir jetzt die Approximationsgüte des von uns vorgestellten Greedy-Algorithmus abschätzen.

Theorem 5.28 *Sei s' der durch den Greedy-Algorithmus konstruierte Superstring für $S = \{s_1, \ldots, s_k\}$. Dann gilt:*

$$|s'| \leq 4 \cdot SSP(S).$$

Beweis: Sei $\tilde{s}_i$ für $i \in [1 : r]$ der jeweils längste String aus C_i in einer optimalen Zyklenüberdeckung $C = \bigcup_{i=1}^{r} C_i$ für G_S. Sei jetzt $\tilde{s}$ ein kürzester Superstring für $\tilde{S} = \{\tilde{s}_1, \ldots, \tilde{s}_k\}$. Nach Lemma 5.7 gibt es eine Permutation $(j_1, \ldots, j_r)$ von $[1 : r]$, so dass sich $\tilde{s}$ schreiben lässt als:

$$\tilde{s} = p(\tilde{s}_{j_1}, \tilde{s}_{j_2}) \cdots p(\tilde{s}_{j_{r-1}}, \tilde{s}_{j_r}) \cdot p(\tilde{s}_{j_r}, \tilde{s}_{j_1}) \cdot o(\tilde{s}_{j_r}, \tilde{s}_{j_1}).$$

Dann gilt:

$$\begin{aligned}
|\tilde{s}| &= \sum_{i=1}^{r} |p(\tilde{s}_{j_i}, \tilde{s}_{j_{(i \bmod r)+1}})| + |\underbrace{o(\tilde{s}_{j_r}, \tilde{s}_{j_1})}_{\geq 0}| \\
&\geq \sum_{i=1}^{r} \left(\underbrace{|\tilde{s}_{j_i}|}_{=\ell_i} - |o(\tilde{s}_{j_i}, \tilde{s}_{j_{(i \bmod r)+1}})| \right) \\
&\quad \text{aufgrund des Overlap-Lemmas gilt:} \\
&\quad |o(\tilde{s}_{j_i}, \tilde{s}_{j_{(i \bmod r)+1}})| \leq (w_{j_i} + w_{j_{(i \bmod r)+1}}) \\
&\geq \sum_{i=1}^{r} \ell_i - \sum_{i=1}^{r} \left(w_{j_i} + w_{j_{(i \bmod r)+1}} \right) \\
&= \sum_{i=1}^{r} \ell_i - \sum_{i=1}^{r} w_{j_i} - \sum_{i=1}^{r} w_{j_{(i \bmod r)+1}} \\
&\quad \text{Verschiebung der Indizes um 1} \\
&= \sum_{i=1}^{r} \ell_i - \sum_{i=1}^{r} w_{j_i} - \sum_{i=1}^{r} w_{j_i} \\
&\quad \text{da } (j_1, \ldots, j_r) \text{ nur eine Permutation von } [1 : r] \text{ ist} \\
&= \sum_{i=1}^{r} \ell_i - 2 \sum_{i=1}^{r} w_i \\
&= \sum_{i=1}^{r} (\ell_i - 2w_i).
\end{aligned}$$

Der kürzeste Superstring für S ist sicherlich nicht kürzer als der für $\tilde{S}$, da ja $\tilde{S} \subseteq S$ gilt. Also gilt:

$$SSP(S) \geq SSP(\tilde{S}) = |\tilde{s}| \geq \sum_{i=1}^{r} (\ell_i - 2w_i). \tag{5.1}$$

Nach Konstruktion des Superstrings s' für s mit Hilfe des Greedy-Algorithmus gilt:

$$\begin{aligned}
|s'| &\leq \sum_{i=1}^{r} (w_i + \ell_i) \\
&\leq \underbrace{\sum_{i=1}^{r} (\ell_i - 2w_i)}_{\leq SSP(S)} + \sum_{i=1}^{r} 3w_i \\
&\quad \text{mit Hilfe von Ungleichung 5.1} \\
&\leq SSP(S) + 3 \cdot SSP(S) \\
&\leq 4 \cdot SSP(S)
\end{aligned}$$

Damit haben wir das Theorem bewiesen. ■

Somit haben wir nachgewiesen, dass der Greedy-Algorithmus eine 4-Approximation für das Shortest Superstring Problem liefert, d.h. der generierte Superstring ist höchstens um den Faktor 4 zu lang.

Wir wollen an dieser Stelle noch anmerken, dass die Aussage, das der Greedy-Algorithmus eine 4-Approximation liefert, nur eine obere Schranke ist. Wir können für den schlimmsten Fall nur beweisen, dass der konstruierte Superstring maximal um den Faktor 4 zu lang ist. Es ist nicht klar, ob die Analyse scharf ist, dass heißt, ob der Algorithmus nicht im worst-case bessere Resultate liefert. Für den average-case können wir davon ausgehen, dass die Ergebnisse besser sind.

Wir können auch eine 3-Approximation beweisen, wenn wir etwas geschickter vorgehen. Nach der Erzeugung einer optimalen Zyklenüberdeckung generieren wir für jeden Zyklus einen Superstring, wie in Korollar 5.6 angegeben. Um den gesamten Superstring zu erhalten, werden die Superstrings für die einzelnen Zyklen einfach aneinander gehängt. Auch hier können wir Überlappungen ausnutzten. Wenn wir dies und noch ein paar weitere kleinere Tricks anwenden, erhalten wir eine 3-Approximation. Der bislang beste bekannte Approximationsalgorithmus für das Shortest Superstring Problem liefert eine 2,5-Approximation.

5.3.6 Zusammenfassung und Beispiel

In Abbildung 5.24 ist die Vorgehensweise für die Konstruktion eines kürzesten Superstrings mit Hilfe der Greedy-Methode noch einemal skizziert.

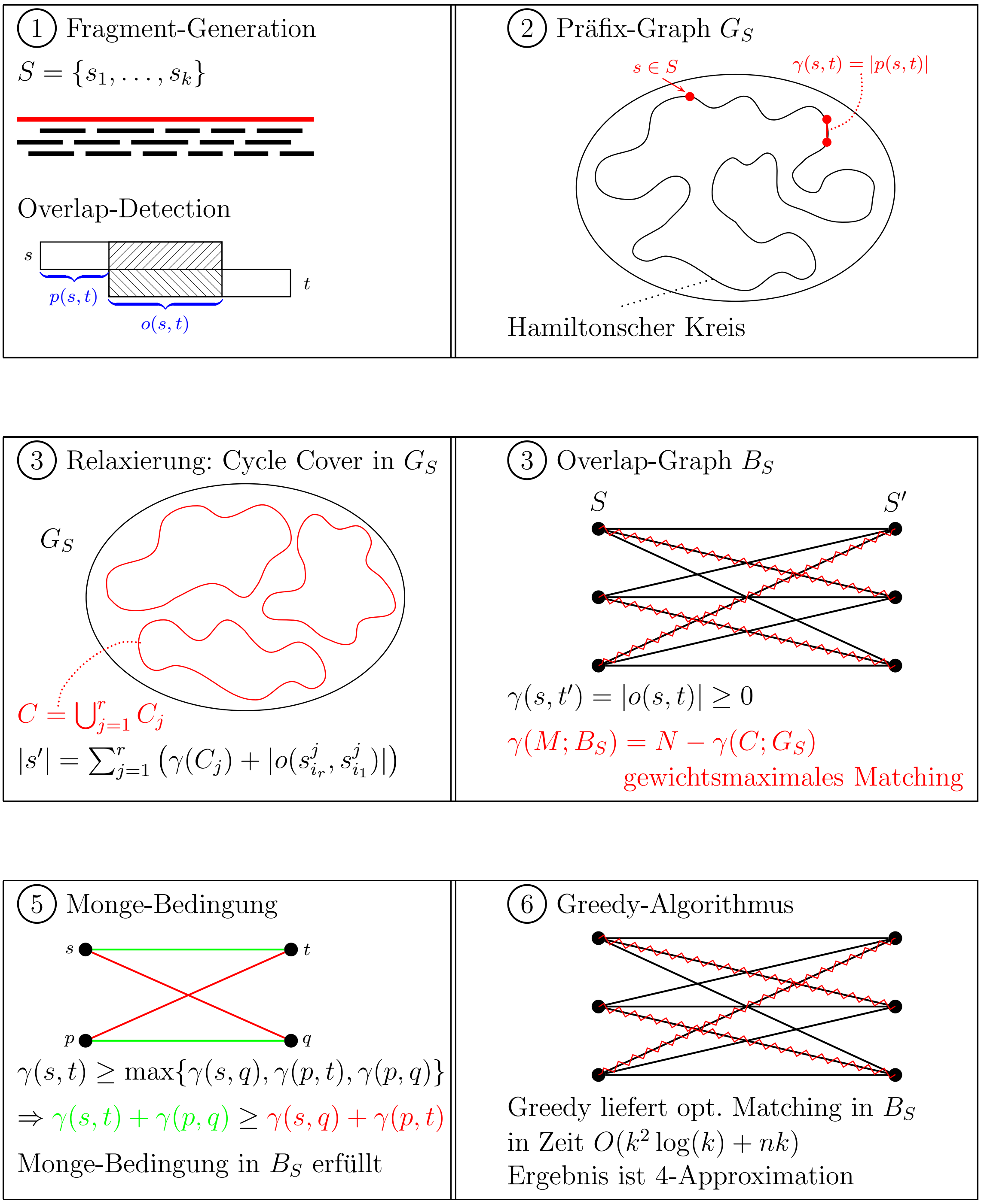

Abbildung 5.24: Skizze: Zusammenfassung der Greedy-SSP-Approximation

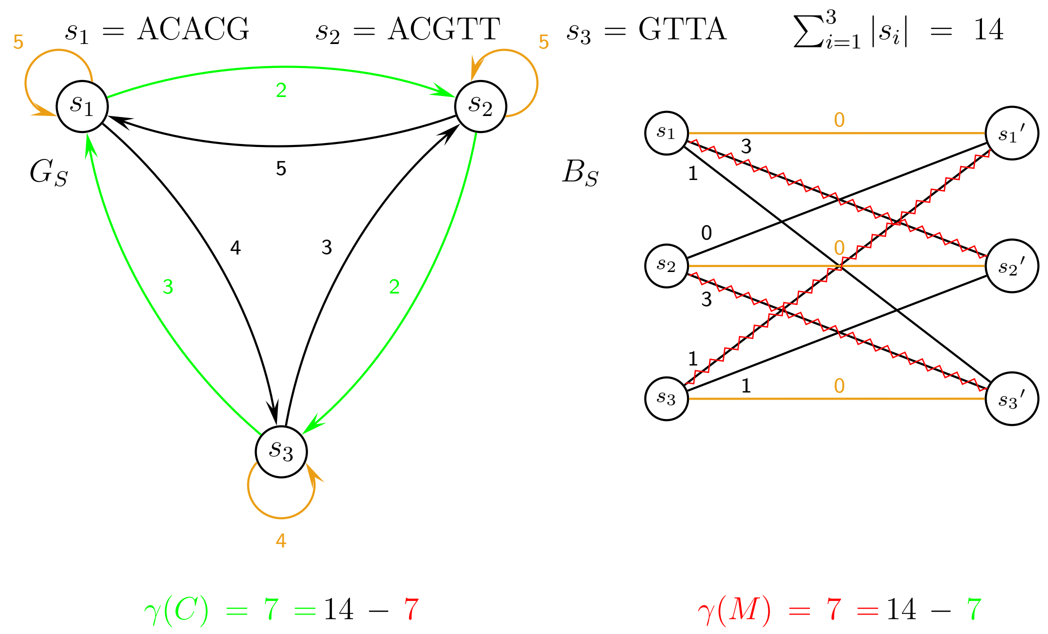

Abbildung 5.25: Beispiel: Greedy-Algorithmus für Superstrings

Zum Abschluss vervollständigen wir unser Beispiel vom Beginn dieses Abschnittes und konstruieren mit dem Greedy-Algorithmus einen kürzesten Superstring, der hier optimal sein wird. In Abbildung 5.25 ist links der zugehörige Präfix-Graph und rechts der zugehörige Overlap-Graph angegeben.

Zuerst bestimmen wir das maximale Matching mit dem Greedy-Algorithmus im Overlap-Graphen. Dazu wird zuerst die Kanten $\{s_1, s_2'\}$ gewählt. Wir hätten auch $\{s_2, s_3'\}$ wählen können. Egal welche hier zuerst gewählt wird, die andere wird als zweite Kante ins Matching aufgenommen. Zum Schluss bleibt nur noch die Kante $\{s_3, s_1'\}$ übrig, die aufzunehmen ist.

Dies entspricht der folgenden Zyklenüberdeckung im zugehörigen Präfix-Graphen (s_1, s_2, s_3) (oder aber (s_2, s_3, s_1) bzw. (s_3, s_1, s_2), je nachdem, wo wir den Kreis bei willkürlich aufbrechen). Wir erhalten hier also sogar einen hamiltonschen Kreis.

Nun müssen wir aus der Zyklenüberdeckung nur noch den Superstring konstruieren. Wie wir gesehen haben, ist es am günstigsten den Kreis nach der Kante aufzubrechen, wo der zugehörige Overlap-Wert klein ist. Daher wählen wir als Kreisdarstellung $(s_1, s_2, s_{,3})$ und erhalten folgenden Superstring:

$$
\begin{array}{cccccccccc}
 & & \mathrm{A} & \mathrm{C} & \mathrm{A} & \mathrm{C} & \mathrm{G} & & & \\
 & & & & \mathrm{A} & \mathrm{C} & \mathrm{G} & \mathrm{T} & \mathrm{T} & \\
 & & & & & & \mathrm{G} & \mathrm{T} & \mathrm{T} & \mathrm{A} \\
\hline
s & = & \mathrm{A} & \mathrm{C} & \mathrm{A} & \mathrm{C} & \mathrm{G} & \mathrm{T} & \mathrm{T} & \mathrm{A}
\end{array}
$$

Mit Bezug zum Präfix-Graphen ergibt sich folgenden Korrespondenz zu den Knoten im hamiltonschen Kreis:

$$s = \underbrace{AC}_{p(s_1,s_2)}\ \underbrace{AC}_{p(s_2,s_3)}\ \underbrace{GTT}_{p(s_3,s_1)}\ \underbrace{A}_{o(s_3,s_1)}\ .$$

5.4 (*) Whole Genome Shotgun-Sequencing

Wie schon angemerkt, wurde vermutet, dass die eben vorgestellte Methode nur für nicht zu lange oder einfachere Genome (ohne Repeats) anwendbar ist. Celera Genomics hat mit der Sequenzierung der Fruchtfliege Drosophila Melanogaster und dem menschlichen Genom bewiesen, dass sich dieses Verfahren prinzipiell auch zur Sequenzierung ganzer Genome anwenden lässt. Natürlich sind hierzu noch ein paar weitere Tricks nötig, auf die wir hier noch ganz kurz eingehen wollen.

5.4.1 Sequencing by Hybridization

Um eine der dabei verwendeten Methode kennen zu lernen, gehen wir noch einmal auf die Methode der Sequenzierung durch Hybridisierung zurück. Hierbei werden mithilfe von DNA-Microarrays alle Teilfolgen einer festen Länge ermittelt, die in der zu sequenzierenden Seqeunz auftreten. Betrachten wir hierzu ein Beispiel dass in Abbildung 5.26 angegeben ist. In unserem Beispiel erhalten wir also die folgende

T	G	A	C	G	A	C	A	G	A	C	T
T	G	A	C								
	G	A	C	G							
		A	C	G	A						
			C	G	A	C					
				G	A	C	A				
					A	C	A	G			
						C	A	G	A		
							A	G	A	C	
								G	A	C	T

Abbildung 5.26: Beispiel: Teilsequenzen der Länge 4, die bei SBH ermittelt werden

Menge an (sehr kurzen, wie für SBH charakteristisch) so genannten *Oligos*:

$$\{\text{ACGA}, \text{ACAG}, \text{AGAC}, \text{CAGA}, \text{CGAC}, \text{GACA}, \text{GACG}, \text{GACT}, \text{TGAC}\}.$$

Auch hier müssen wir wieder einen Superstring für diese Menge konstruieren. Allerdings würden wir mehrfach vorkommenden Teilsequenzen nicht feststellen. Diese Information erhalten wir über unser Experiment erst einmal nicht, so dass wie eine etwas andere Modellierung finden müssen. Außerdem versuchen wie die Zusatzinformation auszunutzen, dass (bei Nichtberücksichtigung von Fehlern) an jeder Position des DNS-Stranges ein Oligo der betrachteten Länge bekannt ist.

Beim SSP haben wir in einem Graphen einen hamiltonschen Kreis gesucht. Dies war ein schwieriges Problem. In der Graphentheorie gibt es ein sehr ähnliches Problem, nämlich das Auffinden eines eulerschen Kreises, was hingegen algorithmisch sehr leicht ist.

Definition 5.29 *Sei $G = (V, E)$ ein gerichteter Graph. Ein Pfad $p = (v_1, \ldots, v_\ell)$ heißt* eulersch, *wenn alle Kanten des Graphen genau einmal in diesem Pfad enthalten sind, d.h.:*

- $(v_{i-1}, v_i) \in E$ *für alle* $i \in [2 : \ell]$,
- $|\{(v_{i-1}, v_i) \, : \, i \in [2 : \ell]\}| = |E|$.

Ein Graph heißt eulersch, *wenn er einen eulerschen Pfad besitzt.*

Wir weisen hier darauf hin, dass wir aus gegebenem Anlass der Begriff eulerscher Graph anders definieren als in der Literatur üblich. Dort wird ein Graph als eulersch definiert, wenn er einen eulerschen Kreis besitzt. Da wir hier aber an einem Pfad als Ergebnis und nicht an einem Kreis interessiert sind, wird der Grund für unsere Definition klar. Wir wiederholen noch kurz das Ergebnis, dass es sich sehr effizient feststellen lässt ob ein Graph eulersch ist bzw. einen eulerschen Pfad enthält.

Lemma 5.30 *Sei $G = (V, E)$ ein gerichteter Graph. Der Graph G ist genau dann eulersch, wenn es zwei Knoten $u, w \in V$ gibt, so dass folgendes gilt:*

- $d^-(v) = d^+(v)$ *für alle* $v \in V \setminus \{u, w\}$,
- $d^-(u) + 1 = d^+(u)$ *und*
- $d^-(w) = d^+(w) + 1$.

Ein eulerscher Pfad in G kann in Zeit $O(|V| + |E|)$ ermittelt werden, sofern ein solcher existiert.

Der Beweis sei dem Leser überlassen bzw. wir verweisen auf die einschlägige Literatur hierfür. Wir werden jetzt sehen, wie wir diese Eigenschaft ausnutzen können. Dazu definieren wir für eine Menge von Oligos einen so genannten

Definition 5.31 *Sei* $S = \{s_1, \ldots, s_k\}$ *eine Menge von* ℓ*-Oligos über* Σ*, d.h.* $|s_i| = \ell$ *für alle* $i \in [1:k]$*. Der gerichtete Graph* $G_S = (V, E)$ *heißt* Oligo-Graph*, wobei*

- $V = \{s_1 \cdots s_{\ell-1}, s_2 \cdots s_\ell \,:\, s \in S\} \subseteq \Sigma^{\ell-1}$,
- $E = \{(v, w) \,:\, \exists s \in S : v = s_1 \cdots s_{\ell-2} \wedge w = s_2 \cdots s_{\ell-1}\}$.

Als Knotenmenge nehmen wir alle $(\ell - 1)$-Tupel aus $\Sigma^{\ell-1}$ her. Damit die Knotenmenge im Zweifelsfall nicht zu groß wird, beschränken wir uns auf alle solchen $(\ell - 1)$-Tupel, die ein Präfix oder Suffix eines Oligos sind. Kanten zwischen zwei solcher $(\ell - 1)$-Tupel führen wir von einem Präfix zu einem Suffix desselben Oligos. Wir wollen uns diese Definition noch an unserem Beispiel in Abbildung 5.27 veranschaulichen. Wie man dem Beispiel ansieht, kann es durchaus mehrere eulersche

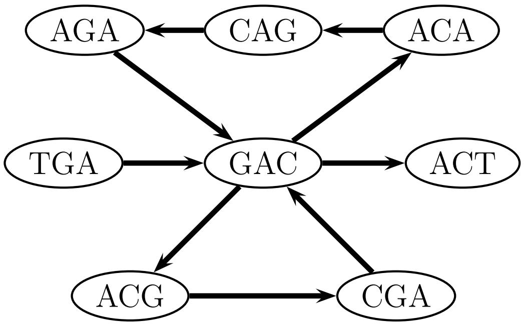

Abbildung 5.27: Beispiel: Oligo-Graph

Pfade im Oligo-Graphen geben. Einer davon entspricht der ursprünglichen Sequenz.

Probleme hierbei stellen natürlich Sequenzierfehler dar, die den gesuchten eulerschen Pfad zerstören können. Ebenso können lange Repeats (größer gleich ℓ) zu Problemen führen. Wäre im obigen Beispiel das letzte Zeichen der Seqeunz ein A, so gäbe es ein Repeat der Länge 4, nämlich GACA. Im Oligo-Graphen würde das dazu führen dass die Knoten ACT und ACA verschmelzen würden. Der Graph hätte dann ebenfalls keinen eulerschen Pfad mehr (außer wir würden Mehrfachkanten erlauben, hier eine Doppelkante zwischen GAC nach ACA).

5.4.2 Anwendung auf Fragment Assembly

Könnte uns die Technik der eulerschen Pfade beim Fragment Assembly helfen? Ja, die Idee ist die Folgende. Wir kennen ja Sequenzen der Länge 500. Diese teilen wir in überlappende Oligos der Länge ℓ (in der Praxis wählt man $\ell \approx 20$) wie folgt ein. Sei $s = s_1 \cdots s_n$ ein Fragment, dann erhalten wir daraus $n - \ell + 1$ ℓ-Oligos durch $s^{(i,\ell)} = s_i \cdots s_{i+\ell-1}$ für $i \in [1 : n - \ell + 1]$.

Diese Idee geht auf Idury und Waterman zurück und funktioniert, wenn es keine Sequenzierfehler und nur kurze Repeats gibt. Natürlich müssen wir auch hier voraussetzen, das die zu sequenzierende Sequenz gut überdeckt ist, dass heißt jedes Nukleotid wird durch mindestens ℓ verschiedene Oligos überdeckt.

Dieser Ansatz hat allerdings auch den Vorteil, dass man versuchen kann die Fehler zu reduzieren. Ein Sequenzierfehler erzeugt genau ℓ fehlerhafte Oligos (außer der Fehler taucht am Rand des Fragments auf, dann natürlich entsprechend weniger). Hierbei nutzt man aus, dass eine Position ja von vielen Fragmenten und somit auch Oligos an derselben Position überdeckt wird (in der Praxis etwa 10) und dass pro Oligo aufgrund deren Kürze (in der Praxis etwa 20) nur wenige Sequenzierfehler (möglichst einer) vorliegen.

Dazu ein paar Definitionen. Ein Oligo heißt *solide*, wenn es in einer bestimmen Mindestanzahl der vorliegenden Fragmente vorkommt (beispielsweise mindestens in der Hälfte). Zwei Oligos heißen *benachbart*, wenn sie durch eine Substitution ineinander überführt werden können. Ein Oligo heißt *Waise*, wenn es nicht solide ist, und es zu genau einem anderen soliden Oligo benachbart ist.

Beim Korrekturvorgang suchen wir nach Waisen und ersetzen diese in den Fragmenten durch ihren soliden Nachbarn. Mithilfe dieser Prozedur kann die Anzahl der Fehler deutlich reduziert werden. Hierbei ist anzumerken, dass Fehler hier nicht bezüglich der korrekten Sequenz gemeint ist, sondern so zu verstehen ist, dass Fehler reduziert werden, die im zugehörigen Oligo-Graphen eulersche Pfade eliminieren.

Wie wir schon vorher kurz angemerkt haben, können Repeats ebenfalls eulersche Pfade eliminieren. Um dies möglichst gering zu halten, erlauben wir in unserem Graphen mehrfache Kanten. Außerdem haben wir in unserem Oligo-Graphen ja noch eine wichtige Zusatzinformation. Die Oligos sind ja nicht durch Hybridisierungsexperimente entstanden, sondern wir haben sie aus den Sequenzinformationen der Fragmente abgelesen. Ein Fragment der Länge n induziert daher nicht nur $n-\ell+1$ Oligos, sondern wir kennen ja auch die Reihenfolge dieser Oligos. Das heißt nichts anderes, als dass jedes Fragment einen Pfad der Länge $n-\ell$ auf den $n-\ell+1$ Oligos induziert. Somit suchen wir jetzt nach einem eulerschen Pfad im Oligo-Graphen, der diese Pfade respektiert. Dies macht die Aufgabe in der Hinsicht leichter, dass bei mehreren möglichen eulerschen Pfaden leichter ersichtlich ist, welche Variante zu wählen ist.

Ein weiterer Trick den Celera Genomics bei der Sequenzierung des menschlichen Genoms angewendet hat ist, dass nicht Fragmente der Länge 500 sequenziert worden sind, sondern dass man hat Bruchstücke der Länge von etwa 2000 und 10000 Basenpaaren konstruiert. Diese werden dann von beiden Seiten her auf 500 Basenpaare ansequenziert. Dies hat den Vorteil, dass man für die meisten sequenzierten Teile (nicht für alle, aufgrund von Sequenzierfehlern) auch noch jeweils ein Geschwister-Teil im Abstand von 2000 bzw. 10000 Basenpaaren kennt. Dies erlaubt beim Zusammensetzen der Teile eine weitere Überprüfung, ob Abstand und Orientierung der Geschwister-Fragmente korrekt sind.

Literaturhinweise

A.1 Lehrbücher zur Vorlesung

Peter Clote, Rolf Backofen: *Introduction to Computational Biology*; John Wiley and Sons, 2000.

Richard Durbin, Sean Eddy, Anders Krogh, Graeme Mitchison: *Biological Sequence Analysis*; Cambridge University Press, 1998.

Dan Gusfield: *Algorithms on Strings, Trees, and Sequences — Computer Science and Computational Biology*; Cambridge University Press, 1997.

David W. Mount: *Bioinformatics — Sequence and Genome Analysis*, Cold Spring Harbor Laboratory Press, 2001.

Pavel A. Pevzner: *Computational Molecular Biology - An Algorithmic Approach*; MIT Press, 2000.

João Carlos Setubal, João Meidanis: *Introduction to Computational Molecular Biology*; PWS Publishing Company, 1997.

Michael S. Waterman: *Introduction to Computational Biology: Maps, Sequences, and Genomes*; Chapman and Hall, 1995.

A.2 Skripten anderer Universitäten

Bonnie Berger: *Introduction to Computational Molecular Biology*, Massachusetts Institute of Technology, http://theory.lcs.mit.edu/~bab/01-18.417-home.html;

Bonnie Berger, *Topics in Computational Molecular Biology*, Massachusetts Institute of Technology, Spring 2001, http://theory.lcs.mit.edu/~bab/01-18.418-home.html;

Paul Fischer: *Einführung in die Bioinformatik* Universität Dortmund, Lehrstuhl II, WS2001/2002, http://ls2-www.cs.uni-dortmund.de/lehre/winter200102/bioinf/

Richard Karp, Larry Ruzzo: *Algorithms in Molecular Biology*; CSE 590BI, University of Washington, Winter 1998. http://www.cs.washington.edu/education/courses/590bi/98wi/

Larry Ruzzo: *Computational Biology*, CSE 527, University of Washington, Fall 2001; http://www.cs.washington.edu/education/courses/527/01au/

Georg Schnittger: *Algorithmen der Bioinformatik*, Johann Wolfgang Goethe-Universität Frankfurt am Main, Theoretische Informatik, WS 2000/2001, http://www.thi.informatik.uni-frankfurt.de/BIO/skript2.ps.

Ron Shamir: *Algorithms in Molecular Biology* Tel Aviv University, http://www.math.tau.ac.il/~rshamir/algmb.html; http://www.math.tau.ac.il/~rshamir/algmb/01/algmb01.html.

Ron Shamir: *Analysis of Gene Expression Data, DNA Chips and Gene Networks*, Tel Aviv University, 2002; http://www.math.tau.ac.il/~rshamir/ge/02/ge02.html;

Martin Tompa: *Computational Biology*, CSE 527, University of Washington, Winter 2000. http://www.cs.washington.edu/education/courses/527/00wi/

A.3 Lehrbücher zu angrenzenden Themen

Teresa K. Attwood, David J. Parry-Smith; *Introduction to Bioinformatics*; Prentice Hall, 1999.

Maxime Crochemore, Wojciech Rytter: *Text Algorithms*; Oxford University Press: New York, Oxford, 1994.

Martin C. Golumbic: *Algorithmic Graph Theory and perfect Graphs*; Academic Press, 1980.

Benjamin Lewin: *Genes*; Oxford University Press, 2000.

Milton B. Ormerod: *Struktur und Eigenschaften chemischer Verbindungen*; Verlag Chemie, 1976.

Hooman H. Rashidi, Lukas K. Bühler: *Grundriss der Bioinformatik — Anwendungen in den Biowissenschaften und der Medizin*,

Klaus Simon: *Effiziente Algorithmen für perfekte Graphen*; Teubner, 1992.

Maxine Singer, Paul Berg: *Gene und Genome*; Spektrum Akademischer Verlag, 2000.

Lubert Stryer: *Biochemie*, Spektrum Akademischer Verlag, 4. Auflage, 1996.

A.4 Originalarbeiten

Kellogg S. Booth, George S. Lueker: Testing for the Consecutive Ones property, Interval Graphs, and Graph Planarity Using PS-Tree Algorithms; *Journal of Computer and System Science*, Vol.13, 335–379, 1976.

Ting Chen, Ming-Yang Kao: On the Informational Asymmetry Between Upper and Lower Bounds for Ultrametric Evolutionary Trees, *Proceedings of the 7th Annual European Symposium on Algorithms, ESA'99*, Lecture Notes in Computer Science 1643, 248–256, Springer-Verlag, 1999.

Richard Cole: Tight Bounds on the Complexity of the Boyer-Moore String Matching Algorithm; *SIAM Journal on Computing*, Vol. 23, No. 5, 1075–1091, 1994.
s.a. *Technical Report*, Department of Computer Science, Courant Institute for Mathematical Sciences, New York University, TR1990-512, June, 1990, http://csdocs.cs.nyu.edu/Dienst/UI/2.0/Describe/ncstrl.nyu_cs%2fTR1990-512

Martin Farach, Sampath Kannan, Tandy Warnow: A Robust Model for Finding Optimal Evolutionary Trees, *Algorithmica*, Vol. 13, 155–179, 1995.

Wen-Lian Hsu: PC-Trees vs. PQ-Trees; *Proceedings of the 7th Annual International Conference on Computing and Combinatorics, COCOON 2001*, Lecture Notes in Computer Science 2108, 207–217, Springer-Verlag, 2001.

Wen-Lian Hsu: A Simple Test for the Consecutive Ones Property; *Journal of Algorithms*, Vol.43, No.1, 1–16, 2002.

Haim Kaplan, Ron Shamir: Bounded Degree Interval Sandwich Problems; *Algorithmica*, Vol. 24, 96–104, 1999.

Edward M. McCreight: A Space-Economical Suffix Tree Construction Algorithm; *Journal of the ACM*, Vol. 23, 262–272,1976.

Moritz Maaß: *Suffix Trees and Their Applications*, Ausarbeitung von der Ferienakademie '99, Kurs 2, Bäume: Algorithmik und Kombinatorik, 1999. http://www14.in.tum.de/konferenzen/Ferienakademie99/

Esko Ukkonen: On-Line Construction of Suffix Tress, *Algorithmica*, Vol. 14, 149–260, 1995.

Index

F

G

H