Michael Austermann 4. Mai 1998

A-Seminar
Bioinformatik

Fragment Assembly (shortest superstring)

0. Abstrakt

1. Einleitung

2. Shotgun DNA-Sequencing

3. Shortest Superstring

4. Zusammenfassung

5. Literatur

0. Abstrakt

Aufgrund von technisch bedingten Einschränkungen ist es nur möglich kleine Teile einer großen DNA-Sequenz zu entschlüsseln, also die Basen zu bestimmen. Werden nun viele Kopien der Ursprungssequenz statistisch geteilt und entschlüsselt, macht man sich die Überlappung einzelner Teilsequenzen zunutzte um auf die Ursprungssequenz zurückzuschließen. Dieses Szenario ist die Motivation für den zweiten, ausführlichen Teil dieser Ausarbeitung, in der man von einer idealisierten Annahme ausgeht. Es werden dann verschiedene Lösungsansätze gezeigt, die den sogenannten Shortest Superstring einer Menge von Strings bis zu einer bestimmten Fehlergrenze effizient approximieren.

1. Einleitung

In der DNA-Forschung ist man an dem Aufbau von DNA-Molekülen interessiert. DNA-Moleküle sind aus 4 verschiedenen Basen (Guanin, Cytosin, Adenin und Thymin) aufgebaut. Die Länge eines DNA-Moleküls kann von einigen wenigen tausend Basen bei einem Virus bis zu einer Größenordnung von 3.000.000.000 bei einem menschlichen Organismus variieren. Also sind auch im Verhältnis kleine Teilstücke immer noch von beträchtlicher Länge.

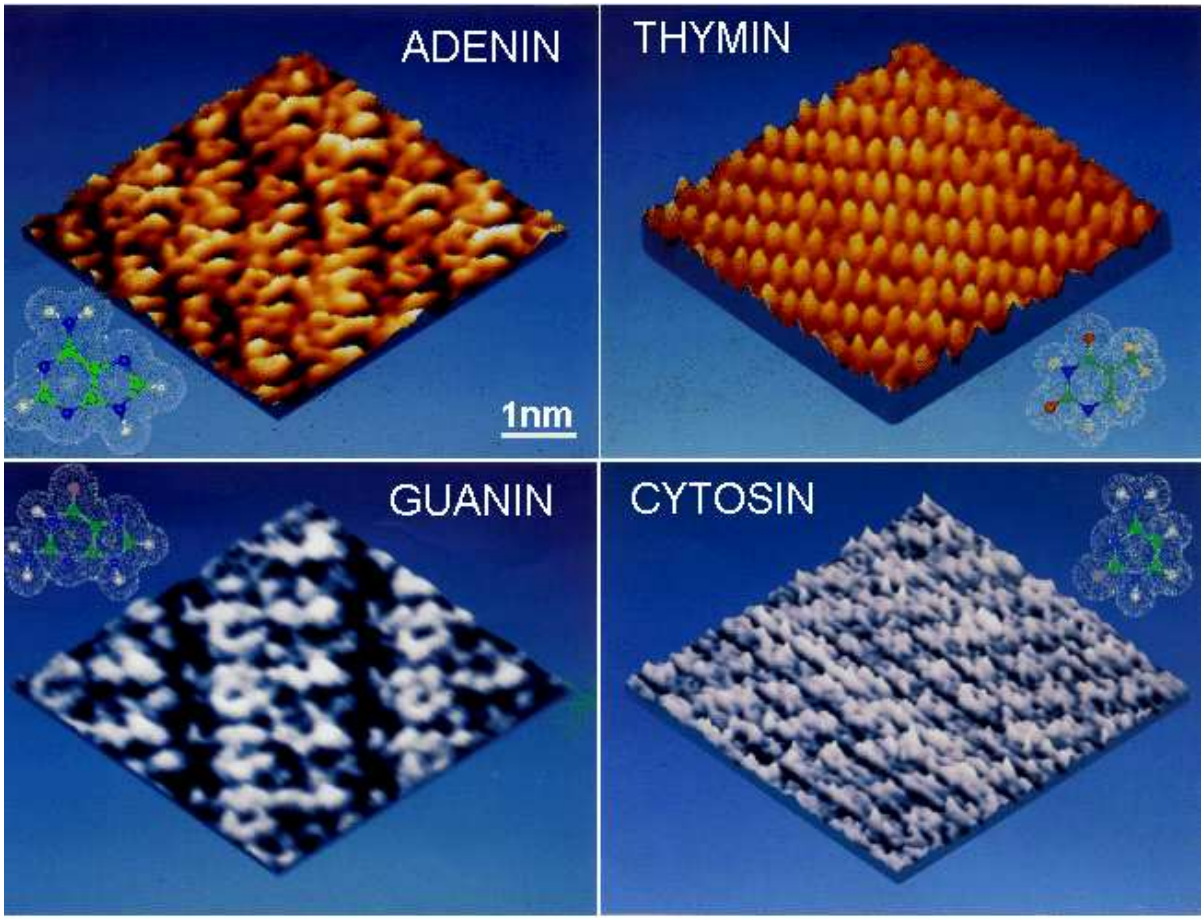

Abbildung 1.1 Adenin, Guanin, Thymin und Cytosin unter dem Rastersondenmikroskop. (Quelle: http://www.kri.physik.uni-muenchen.de/crystal/stm/dna_de.htm)

Man ist heute gerade soweit etwa die ersten 400 bis 500 Basen einer DNA-Sequenz auf einmal lesen zu können. Will man nun die Basen einer großen DNA-Sequenz bestimmen,

muß diese erst durch chemische, enzymatische oder physikalische Mittel in genügend kleine Teile geschnitten werden. Danach können diese Teile dann einzeln gelesen werden. Beim Aufteilen der DNA-Sequenz gehen jedoch die Informationen verloren, in welcher Reihenfolge die Fragmente zusammengesetzt werden müssen um wieder die ursprünglichen DNA-Sequenz zu erhalten. Nun hilft man sich mit einem kleinen Trick. Die ursprüngliche DNA-Kette wird genügend oft kopiert und diese Kopien werden dann in die eben erwähnten lesbaren kleineren Teile aufgeteilt. Beim Zusammensetzten kann man sich nun die Überlappung der einzelnen Teile zur Hilfe nehmen, denn alle Teile stammen schließlich aus den Kopien der selben Ursprungs-DNA-Sequenz, die alle an zufälligen Stellen geteilt wurden.
Es wurden probabilistische Analysen gemacht [5], die gezeigt haben, daß bei einer Ursprungslänge L der DNA-Sequenz und der Annahme, daß alle Teilstücke die Länge 400 haben, etwa 5L/400 bis 10L/400 Teilstücke vorhanden sein müssen um eine gute Rekonstruktion des Urspungs-DNA-Moleküls zu erhalten. Der Vorgang des Zusammenfügens dieser kleinen Teilstücke ist sehr mühsam und soll durch eine Maschine erledigt werden. Ich werde nun im folgenden einen praktischen (in der Wissenschaft benutzten) und drei theoretische Algorithmen, die von idealisierten aber ähnlichen Vorbedingungen ausgehen, vorstellen.

2. Shotgun DNA sequencing

2.1 Sequence assembly

Im folgenden gehe ich davon aus, daß alle DNA-Sequenzen die Länge 400 haben. Des weiteren spreche ich meist nicht mehr von DNA-Sequenzen bzw. Fragmenten sondern von Strings, nämlich Strings über einem 4-elementigen Alphabet, wobei jeder Buchstabe des Alphabets für eine Base steht, d.h. jedes DNA-Molekül wird als Zeichenkette über den Zeichen {A,C,G,T} repräsentiert. Außerdem soll es keine doppelten Strings bzw. Fragmente geben.
Die meisten sequence assembly Methoden beruhen auf drei Schritten, overlap detection, fragment layout, deciding the consensus, so auch der folgende Algorithmus:

2.1.1 Schritt eins: overlap detection

Der erste Schritt in der sequence assembly Methode besteht daraus, für jedes geordnete Paar aus Teilstrings zu bestimmen wie gut das Ende (Suffix) des ersten zum Anfang (Präfix) des zweiten Fragments „paßt“, also zu bestimmen wie stark diese zwei Strings sich überlappen. Hierbei muß man beachten, daß beim Bestimmen der Basen der Teilsequenzen Meßfehler im Bereich von 1% bis 5% auftreten, d.h. man kann nicht einfach den einen String über den anderen legen und prüfen, ob die Überlappung der beiden genau übereinstimmt. Vielmehr sucht man ein Suffix-Präfix-Paar der beiden Strings, dessen Gleichheit maximal ist im Vergleich zu allen möglichen Suffix-Präfix-Paaren. Gleichheit ist hier wie folgt definiert: Überlappt man das Suffix-Präfix-Paar, gibt es für jede übereinstimmende Stelle einen positiven Beitrag und für jede abweichende Stelle einen negativen Beitrag zur Gesamtsumme, wobei die positiven und negativen Beiträge noch geeignet gewichtet werden können. Die Frage ist nun was für Gewichte man verwenden soll, worauf ich hier nicht weiter eingehen möchte. Die Implementation eines Algorithmus zur Bestimmung solcher Suffix-Präfix-Paare sollte klar sein (dynamische Programmierung). Nimmt man an, daß jeder String in etwa die Länge n hat, braucht man also etwa $O(n^2)$ Zeit für die Berechnung des längsten Suffix-Präfix-Paares zweier Strings. Sind insgesamt k Strings vorhanden, dann geht die Berechnung aller möglichen k(k-1) Paare also in $O(k^2n^2)$.
Dieser erste Schritt ist im Vergleich zu den 2 folgenden Schritten der zeitaufwendigste und es gibt daher verschiedene Ansätze ihn zu beschleunigen. Eine Möglichkeit ist zum Beispiel unattraktive Paare die sich stark unterscheiden gar nicht im obigen Schritt zu betrachten, sondern sie auszuschließen. Dann stellt sich aber die Frage wie solche unattraktiven Paare

effizient identifiziert werden können. Dazu kann man nun feststellen, daß zwei Strings nur dann eine ausreichend große Überlappung haben, wenn sie ein genügend großes Teilstück gemeinsam haben. Was nun ein genügend großes Teilstück ist, kann probabilistisch bestimmt werden. Es gibt z.B. einen Algorithmus [1] der für k Strings, jeweils mit Länge n, für jedes der möglichen $\binom{2}{k}$ Paare den längsten gemeinsamen Teilstring in $O(k^2n)$ berechnet.
Man kann also durch einen solchen Algorithmus eine Vorauswahl von String-Paaren treffen, die „attraktiv" genug sind, sie wie oben beschrieben auf ihr längstes Suffix-Präfix-Paar hin zu untersuchen. Diese Beschleunigung des ersten Schrittes hat nur eine geringe Verschlechterung der Qualität des Endergebnisses zur Folge.

2.1.2 Schritt zwei: substring layout

In diesem Schritt versucht man nun die Paare aus Schritt 1 miteinander in Verbindung zu bringen. Aus dem ersten Schritt haben wir nun eine geordnete Liste von String-Paaren, wobei die zwei Strings mit dem besten Suffix-Präfix-Paar ganz oben in der Liste stehen sollen, die zwei zweitbesten danach, usw.. Nun muß man sich das ganze wie folgt vorstellen: Man nimmt das String-Paar mit der besten Überlappung und legt es auf einen imaginären großen „Tisch" (Abbildung 2.1 a). Dann nimmt man das zweitbeste Paar und prüft, ob das erste (schon auf dem „Tisch" liegende) und das zweite Paar einen String gemeinsam haben. Gibt es einen gemeinsamen String legt man den nichtgemeinsamen String des zweiten Paares an der richtigen Stelle an das erste Paar an (Abbildung 2.1 b). Haben die beiden Paare keinen String gemeinsam, legen wir das zweite Paar neben das erste Paar, wobei wir eine Lücke lassen (Abbildung 2.1 c). Dieser Vorgang wird nun solange für das jeweils beste übriggebliebene Paar unserer Liste aus Schritt eins fortgeführt, bis jedes der k Strings auf dem „Tisch" liegt.
Alle Entscheidungen in diesem Schritt werden nur mit Hilfe der Informationen aus dem ersten Schritt heraus getroffen. Eventuelle Inkonsistenzen, also Lücken und Abweichungen der einzelnen Sequenzen untereinander werden hier noch nicht beachtet sondern müssen im Schritt drei behandelt werden. Dieser zweite Schritt geht in der Praxis recht schnell.

a) **abcdaaabbcddeeeab**
bcddeeeabcccddaab

b) **abcdaaabbcddeeeab** **daabbbbbbccdddaaa**
bcddeeeabcccddaab

c) **abcdaaabbcddeeeab** **abcabcdddabc**
bcddeeeabcccddaab **dddabccccaaa**

Abbildung 2.1 Die zwei Möglichkeiten für das Zusammenfügen zweier Paare: a) 1. Paar auf dem „Tisch". b) 1. und 2. Paar haben einen String gemeinsam. c) 1. und 2. Paar haben keinen String gemeinsam.

2.1.3 Schritt drei: deciding the consensus

Das Layout aus Schritt zwei wird nun dazu genutzt einen einzigen großen String aus den k Teilstrings zu konstruieren, d.h. an jeder Stelle der Ergebnis-DNA-Kette muß nun **eine** Base bestimmt werden.

cccccabccbcdddabcccccaab
deeebbcccddaacccccab **dabccccaab**
abcdaaabbcddeeeab **abcabcdddabc**
bcddeeeabcccddaab **dddabccccaaa**

Abbildung 2.2 Beispiel für eine Anordnung nach Schritt 2.

Wie aus Abbildung 2.2 ersichtlich, bekommen wir aus Schritt 2 eine Art (k x ?)-Matrix in der nicht in jeder Zelle ein Eintrag steht. Man muß nun für jede Spalte der Matrix ein Zeichen bzw. eine Base bestimmen. Dieses Zeichen wird dann in unserem Ergebnisstring an der entsprechenden Position eingetragen, bzw. repräsentiert die entsprechende Base in unserer DNA-Kette. Stehen in einer Spalte nur gleiche Zeichen ist das Ergebnis klar. Gibt es hingegen Abweichungen muß entschieden werden, welches Zeichen nun „richtiger" als die anderen ist. Auch hier gibt es wieder verschiedene Ansätze zur Lösung dieses Problems. Der einfachste ist sicherlich die Häufigkeit der einzelnen Zeichen pro Spalte festzustellen und dann das Zeichen mit der höchsten Anzahl zu nehmen. Eine Verfeinerung dieses Verfahrens ist nun, daß die Häufigkeit dieses Zeichens im Vergleich zu den anderen in dieser Spalte über einem bestimmten Grenzwert liegen muß. Sollte dieser Wert nicht erreicht werden, ist der Fehler zu groß und diese Stelle nicht klar bestimmbar (und kann dann z.B. offen gelassen werden).
Werden nun auch umliegende Positionen mit in diese Betrachtung eingezogen, können sogenannte *Fenster* mit zu hoher Abweichung bestimmt werden, die dann auf eine andere Art bestimmt werden müssen, z.B. durch multiple alignment.

2.2 Probleme und andere Lösungsansätze

Ein Problem bei der eben beschriebenen Methode liegt im häufigen Auftreten von Wiederholungen einzelner Teilsequenzen im DNA-Code hochentwickelter Organismen. Dies wird besonders problematisch, wenn diese Teilsequenzen länger als die maximale Leselänge (also z.Z. etwa 400-500) sind. Dadurch kann es passieren, daß die vom obigen Algorithmus berechnete DNA-Sequenz kürzer als die Ursprungssequenz ist. Solche gleichartigen Teilsequenzen werden dann nicht konkateniert, sondern, übereinandergelegt, was dazu führt, daß die berechnete Sequenz kürzer als die Ursprungsequenz ist. Ein weiteres Indiz für häufige Wiederholungen sind Regionen im Layout von Schritt zwei, in denen, im Vergleich zu anderen Regionen, verhältnismäßig viele Stücke „übereinandergelegt" wurden.
Um bessere Ergebnisse zu erzielen gibt es Versuche, Seiteninformationen mit in die Berechnung einzubeziehen. Seiteninformation soll bedeuten, daß man z.B. weiß, daß ein bestimmtes Paar überlappt und ein anderes auf keinen Fall. Weitere Informationen dazu in [4].

3 Das kürzeste Superstring Problem

Nachdem ich einen Algorithmus vorgestellt habe der wirklich in der Molekular-Biologie zum Einsatz kommt, gehe ich nun zu einem abstrakteren Thema über.

3.1 Einführung

Definition Gegeben sei eine Menge von Strings $P=\{S_1,S_2,...,S_k\}$, ein S*uperstring* der Menge P ist ein einzelner String der jeden String aus P als Teilstring enthält.

Beispiel Sei P={abcc,efab,bccla}, dann ist ein trivialer Superstring dieser Menge die einfache Konkatenation der Elemente aus P, also abccefabbccla. efabccla ist ebenfalls ein Superstring der Menge P, jedoch kürzer als der triviale. Im folgenden sind wir daran interessiert, Superstrings zu finden, deren Länge möglichst klein ist.

Definition Für eine Menge P sei $S^*(P)$ der kürzeste Superstring von P. Wenn P aus dem Kontext heraus klar ist, werde ich nur S* schreiben.

Das Problem einen kürzesten Superstring für eine Menge von Strings zu finden, hat hauptsächlich zwei Anwendungsgebiete. Erstens ist es eine idealisierte Form unseres oben behandelten Sequence Assembly Problems. Es ist jedoch ein schlechtes Model für das Zusammenfügen von DNA-Fragmenten, da die gemessenen Daten bekannterweise Meßfehler enthalten und diese Fehler in der Theorie der kürzesten Superstrings nicht berücksichtigt werden. Ein weiteres Anwendungsgebiet ist die Datenkompression. Dazu muß man sich vorstellen, daß man eine Menge P von Strings hat, die, aus welchem Grund auch immer, komprimiert werden soll. Eine Möglichkeit ist dann, den kürzesten Superstring S* zu berechnen und dann für jeden String p aus P nur noch die Startposition in S* und die Länge von p anzugeben.

Einen kürzesten Superstring zu finden ist ein NP-hartes Problem, d.h. es ist unwahrscheinlich, daß man einen effizienten Algorithmus findet, der das Optimum in polynomieller Zeit berechnet. Darüber hinaus ist das kürzeste Superstring Problem sogar MAX-SNP-hart. Daher ist es ebenfalls unwahrscheinlich, einen Algorithmus mit polynomieller Laufzeit zu finden, der ein Ergebnis liefert, daß das Optimum bis zu einem beliebigen vorher festgesetzten Faktor approximiert. Hingegen sind Methoden bekannt, die das Optimum bis zu einer bestimmten oberen Schranke approximieren. Im folgenden werden wir uns mit zwei Algorithmen beschäftigen, die einmal einen Superstring berechnen, der maximal 4 mal so lang wie das Optimum ist und im zweiten Fall maximal 3 mal so lang. Das sind jedoch obere Schranken, und die Praxis zeigt, daß die Ergebnisse meist wesentlich besser sind.

3.2 Definitionen

Im weiteren Verlauf wird angenommen, daß kein String aus P Teilstring eines anderen Strings aus P ist. Solche Teilstrings können effizient identifiziert und entfernt werden. Enthalte die Menge P‘ die Strings aus P ohne Teilstrings, dann ist klar, daß S*(P)=S*(P‘). Es gibt also keinen Verlust wenn diese Annahme gemacht wird.

Definition Für einen beliebigen String S, sei |S| die Länge von S. Für eine Menge P von Strings bezeichne ||P|| die Summe der Längen aller Strings aus P.

Definition Für zwei Strings S_i, S_j, sei *ov*(S_i,S_j) die Länge der größten Überlappung eines Suffix von S_i und eines Präfix von S_j. Sei *p*(S_i,S_j) := |S_i| - *ov*(S_i,S_j) und *pref*(S_i,S_j) sei Präfix von S_i mit der Länge *p*(S_i,S_j).

Definition Sei L=o_1,o_2,...,o_t eine geordnete Teilmenge der Zahlen von 1 bis k, wobei k=|P|. L definiert dann eine Teilmenge $\{S_{o_1}, S_{o_2}, ..., S_{o_t}\} \subseteq P$ mit t Strings aus P. Außerdem definiert es den folgenden String:

$$S(L) = pref(S_{o_1}, S_{o_2})\, pref(S_{o_2}, S_{o_3}) ... pref(S_{o_{t-1}}, S_{o_t})\, S_{o_t}$$

Beispiel S_1=azwad, S_2=adcste, S_3=stee, S_4=eeaz, dann folgt mit L=1,2,3,4 S(L)=azwadcsteeaz.

Es ergeben sich nun zwei wichtige Folgerungen für den String S(L).

Lemma 1 Die Länge von String S(L) ist $\sum_{i=1}^{t-1} p(S_{o_i}, S_{o_{i+1}}) + |S_{o_t}|$.

Beweis $S(L) = pref(S_{o_1}, S_{o_2})\, pref(S_{o_2}, S_{o_3}) ... pref(S_{o_{t-1}}, S_{o_t})\, S_{o_t}$, also

$$|S(L)| = |\, pref(S_{o_1}, S_{o_2})\, pref(S_{o_2}, S_{o_3}) ... pref(S_{o_{t-1}}, S_{o_t})\, S_{o_t}\,|$$

$$\Leftrightarrow \quad |S(L)| = |pref(S_{o_1}, S_{o_2})| + |pref(S_{o_2}, S_{o_3})| + ... + |pref(S_{o_{t-1}}, S_{o_t})| + |S_{o_t}|$$

$$\Leftrightarrow \quad |S(L)| = p(S_{o_1}, S_{o_2}) + p(S_{o_2}, S_{o_3}) + ... + p(S_{o_{t-1}}, S_{o_t}) + |S_{o_t}|$$

$$\Leftrightarrow \quad |S(L)| = \sum_{i=1}^{t-1} p(S_{o_{i1}}, S_{o_{i+1}}) + |S_{o_t}|.$$

Lemma 2 Seien L und S(L) wie oben definiert. Dann ist S(L) ein Superstring (nicht unbedingt der kürzeste) der Strings $\{S_{o_1}, S_{o_2}, ..., S_{o_t}\}$.

Beweis durch vollständige Induktion über i.

z.z. Für alle σ_i mit $1 \le i \le t$ gilt: $S_{\sigma i}$ ist Teilstring von S(L).

Induktionsverankerung für i = t:
$S_{\sigma t}$ ist Teilstring von S(L). √ Folgt aus der Definition für S(L)

Induktionsschluß von i auf i-1:
Ist $S_{\sigma i}$ Teilstring von S(L), dann ist auch $S_{\sigma i-1}$ Teilstring von S(L), denn nach Definition enthält *pref*(S_{oi-1},S_{oi}) einen Präfix von S_{oi-1} und S_{oi} den gesamten Rest, denn nach Voraussetzung ist **kein** String aus P Teilstring eines anderen.

Wenn nun L eine Permutation der Zahlen 1 bis k ist, dann impliziert Lemma 2 folgende Aussage:
Korollar 1 Sei P eine Menge von k Strings. Jede Permutation Π der Zahlen 1 bis k spezifiziert einen Superstring S(Π) von P.

Beweis Dieses Korollar folgt sofort aus Lemma 2 mit t = k.

Wenn nun S* ein kürzester Superstring der Menge P ist, dann existiert eine Permutation Π, so daß S* = S(Π).

Beispiel Sei die Menge P={S_1=eeaz, S_2=adcste, S_3=azwad, S_4=stee} mit S*= azwadcsteeaz. Dann ist Π = 3,2,4,1.

Aus Korollar 1 folgt nun mit der Permutation $\Pi = \sigma_1,\sigma_2,...,\sigma_k$ für die Länge von S(Π):

$$S(\Pi) = \sum_{i=1}^{k-1} p(S_{o_i}, S_{o_{i+1}}) + |S_{o_k}|.$$

Hier wird ersichtlich, daß das kürzeste Superstring Problem ein typisches Optimierungsproblem ist, denn der kürzeste Superstring ist dadurch definiert, daß man eine Permutation $\Pi = \sigma_1,\sigma_2,...,\sigma k$ sucht, die den folgen Ausdruck minimiert:

$$\sum_{i=1}^{k-1} p(S_{o_i}, S_{o_{i+1}}) + |S_{o_k}|.$$

Mit $p(S_{o_i}, S_{o_{i+1}}) = |S_{o_i}| - ov(S_{o_i}, S_{o_{i+1}})$ folgt dann:

$$|S(\Pi)| = \sum_{i=1}^{k-1} p(S_{o_i}, S_{o_{i+1}}) + |S_{o_k}| = \sum_{i=1}^{k} |S_{o_i}| - \sum_{i=1}^{k-1} ov(S_{o_i}, S_{o_{i+1}}) = \|P\| - \sum_{i=1}^{k-1} ov(S_{o_i}, S_{o_{i+1}}).$$

Der kürzeste Superstring kann also auch so definiert werden, daß man die Permutation sucht, die $\sum_{i=1}^{k-1} ov(S_{o_i}, S_{o_{i+1}})$ maximiert.

3.3 GREEDY-Algorithmus (kurz)

An dieser Stelle möchte ich einen kleinen Einschub machen und kurz den sogenannten Greedy-Algorithmus vorstellen. Aus der obigen Definition für einen kürzesten Superstring stammte wohl die Idee für diesen Algorithmus: Es werden immer die zwei Strings miteinander vermischt, die im Vergleich zu allen anderen Paaren die maximale Überlappung haben, bis nur noch ein String übrig ist.

Sei P die Menge der Eingabe-Strings.

```
S := P;
while |S|>1 do
{
        seien si,sj zwei Strings aus S mit maximaler Überlappung, dann
        S := S / {si,sj} ∪ pref(si,sj)sj.
}
```

Sei s der verbliebene String aus S, dann ist s Superstring der Menge P und $|s| \leq 4*|S^*|$, soll heißen, daß s maximal die vierfache Länge des Optimums also des kürzesten Superstrings hat. Ich gehe in 3.10 weiter auf eine modifizierte Form (MGreedy) dieses Algorithmus ein.

3.4 Parallelen zu anderen Optimierungsproblemen

Betrachtet man die Definition des kürzesten Superstrings als Minimierungs- bzw. Maximierungsproblem über den Permutationen Π, fällt einem die Ähnlichkeit zum Traveling-Salesman-Problem (TSP) auf. Beim TSP sucht man ebenfalls eine Permutation der Zielpunkte, die die Wegstrecke minimiert. Es ist möglich ein SS-Problem auf ein TS-Problem zu reduzieren. Abbildung 3.2 zeigt die Parallelen zwischen TSP und SSP.

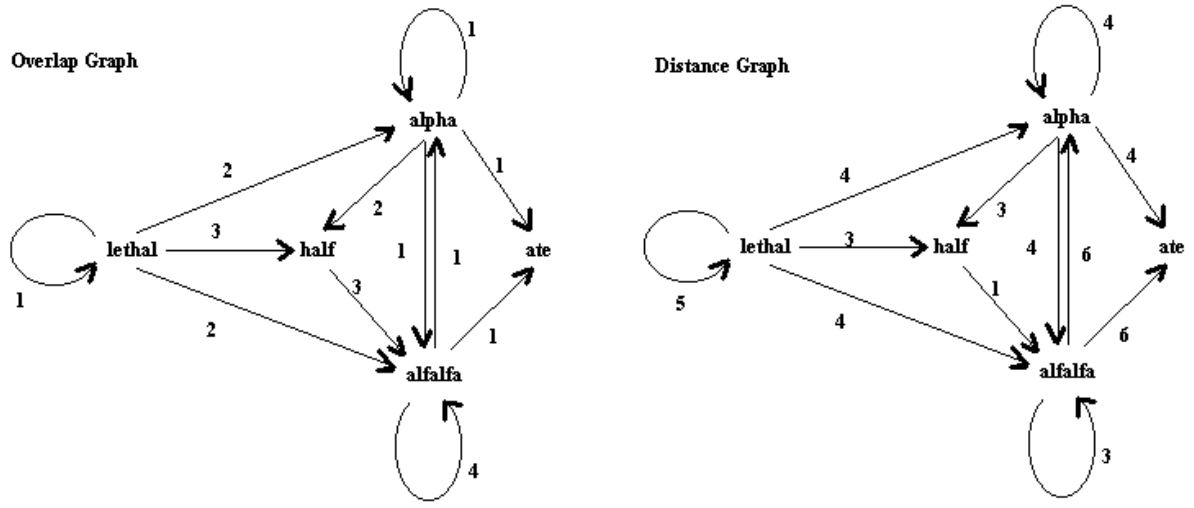

Abbildung 3.2 Überlappungsgraph und Distanzgraph für dieselbe Eingabe.

3.5 Cyclic Strings und Cycle Covers

Definition Gegeben sei ein linearer String S. Der cyclic String ϕ(S) ist eine Kopie von S mit der Veränderung, daß auf das letzte Zeichen von S wieder das erste Zeichen von S folgen soll, ϕ(S) bildet also einen Ring, indem man den Anfang und das Ende von S zusammensetzt.

Definition Die *Länge* |ϕ(S)| des cyclic Strings ϕ(S) ist gleich der Länge |S| von S.

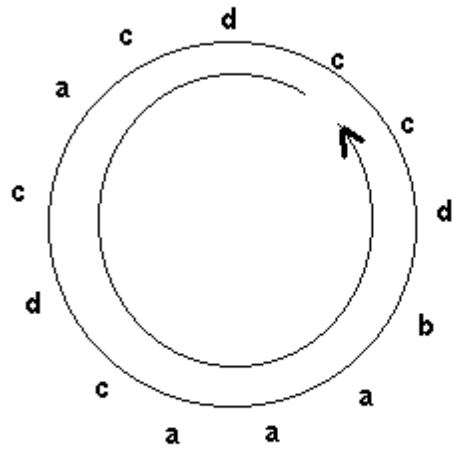

Abbildung 3.3 Beispiel für einen String (unten) und den entsprechenden cyclic String (oben).

Definition Sei *inf*(S) eine unendliche Wiederholung des Strings S. Man sagt, daß ein String α zum cyclic String ϕ(S) *paßt*, wenn α ein Teilstring von *inf*(S) ist. Wenn also α ein Teilstring von *inf*(S) ist, dann beginnt α in *inf*(S) an einer Position zwischen 1 und |S|.

Beispiel Sei S = abac, dann paßt der String bacabacabacab zu ϕ(S).

Definition Ein *Cycle Cover* C(P) von P ist eine Menge von cyclic Strings, so daß jeder String aus P mindestens zu einem cyclic String aus C(P) paßt. Wenn P aus dem Kontext heraus klar ist, schreibe ich im folgenden nur C.

Definition Die *Länge* ||C(P)|| eines Cycle Covers C(P) ist gleich der Summe der Längen aller cyclic Strings aus C(P). Also $\|C\| = \sum_{\Phi \in C} |\Phi|$.

Definition C*(P) bezeichne den Cycle Cover minimaler Länge über der Menge P.

Ein Superstring S von P definiert einen trivialen Cycle Cover über P mit nur einem Element ϕ(S) und der Länge ||C|| gleich |S|. Es folgt,

Lemma 3 Die Länge des kürzesten Superstrings S* der Menge P ist wenigstens ||C*(P)||.

Beweis Ist der kürzeste Superstring S* einer Menge P bekannt, kann man daraus einen trivialen Cycle Cover C der Menge P konstruieren, der die Eigenschaft ||C|| ist gleich ||S*|| hat. C ist dann entweder kleinster Cycle Cover über P oder es existiert ein Cycle Cover C' mit kleinerer Gesamtlänge aller enthaltenen cyclic Strings, also ||C'|| < ||C||. Daraus muß dann Lemma 3 folgen, denn wenn S* bekannt ist findet man **immer** einen Cycle Cover der

maximal dieselbe Länge wie S* hat.

In 3.9 werde ich ein Verfahren angeben mit dem ein Cycle Cover minimaler Länge effizient berechnet werden kann. Wir versuchen deshalb nun eine Methode zu finden, die einen
(kurzen) Superstring aus einem Cycle Cover berechnet.

3.6 Wie bekommt man einen Superstring aus einem Cycle Cover?

Ist nun eine Cycle Cover C(P) für die Menge P gegeben, dann müssen wir die Strings aus P in Teilmengen aufteilen, so daß alle Strings einer Teilmenge zu einem cyclic String aus C(P) passen. Paßt nun ein String aus P zu mehreren cyclic Strings aus C(P) kann eine willkürliche Wahl getroffen werden.

Definition Für einen cyclic String ϕ aus C*(P) bezeichne P_ϕ die Teilmenge der Strings von P die mit ϕ assoziiert sind. Sei im folgenden $|P_\phi| = t$.

Seien nun für jeden cyclic String $\phi(S)$ die Zeichen in ϕ ab dem Beginn durchnummeriert. Dann passen alle Strings aus P_ϕ zu ϕ und starten an einer Position zwischen 1 und |S|. Weil kein String aus P Teilstring eines anderen ist, haben keine zwei Strings aus P_ϕ dieselbe Startposition in ϕ.

Definition Sei $L_\phi=\sigma_1,\sigma_2,...,\sigma_t$ die Liste der **Indizes** aller Strings aus $P\phi$, geordnet nach ihrer Startposition in ϕ. Diese Liste bitte nicht mit einer Reihe von Startpositionen in ϕ verwechseln.

Nach Lemma 2 ist $S(L_\Phi) = pref(S_{o_1}, S_{o_2})\, pref(S_{o_2}, S_{o_3})...pref(S_{o_{t-1}}, S_{o_t})S_{o_t}$ ein Superstring der Menge P_ϕ. Es folgt nun eine interessante Feststellung über die Länge von $S(L_\phi)$, die bei Rotation der Einträge von L_ϕ variiert. Wenn L'_ϕ eine Kreisrotation von L_ϕ ist, dann ist auch $S(L'_\phi)$ ein Superstring, dessen Länge sich jedoch von der von $S(L_\phi)$ unterscheiden kann. Ein Beispiel:
Sei S=azwadcstee. $\phi(S)$ hat dann das Aussehen von S wobei man den ersten mit dem letzten Zeichen verbinden muß. Jeder String der Menge $P=\{S_1,S_2,S_3,S_4\}$ mit S_1=eeaz, S_2=adcste, S_3=azwad, S_4=stee paßt zu $\phi(S)$. Die Reihenfolge ihrer Startpositionen in $\phi(S)$ ist 3,2,4,1.
Sei also L=3,2,4,1 dann hat $S(L_\phi)$= azwadcsteeaz die Länge 12. Rotiert man nun L_ϕ zu L'_ϕ mit L'_ϕ=4,1,3,2 dann ist auch $S(L'_\phi)$=steeazwadcste ein Superstring von P, seine Länge ist jedoch um eins länger als die von $S(L_\phi)$. D.h. wir suchen ein L_ϕ, das die Länge von $S(L_\phi)$ minimiert.

Definition Ist σ_i der letzte Index in der Liste L'_ϕ, dann heißt der String S_{σ_i} der *finale String* von $S(L'_\phi)$.

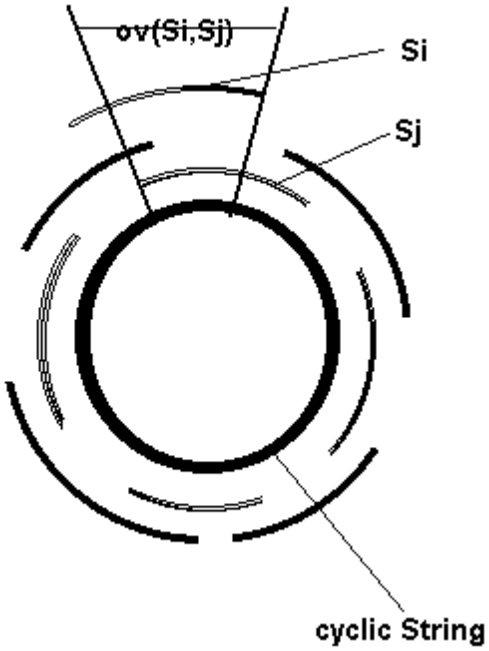

Abbildung 3.4 Ein cyclic String mit *passenden* Strings aus S und „Überhang“ *ov*(S_i,S_j).

Lemma 4 Wenn S_{σ_i} der finale String von S(L'$_\phi$) ist, dann ist die Länge von S(L'$_\phi$) genau $|\Phi| + ov(S_{\sigma_i}, S_{\sigma_{i+1}}) \leq |\Phi| + |S_{\sigma_i}|$, wobei t+1 gleich 1 gesetzt wird.

Dieses Lemma möchte ich nur kurz erläutern: Alle Strings die mit ϕ assoziiert werden passen zu ϕ, d.h. man kann sie um ϕ „herumlegen“. Werden nun alle Strings wie in Abbildung 3.4, beginnend mit $S_{\sigma i+1}$ (in Abbildung 3.4 S_j) in der Reihenfolge von L'$_\phi$ um ϕ gelegt, dann ist $S_{\sigma i+1}$ (in Abbildung 3.4 S_i) der einzige String der nach der einmaligen Umrundung von ϕ „überhängt“. Die Größe dieses Überhangs ist genau *ov*($S_{\sigma i}$,$S_{\sigma i+1}$).

Nach Lemma 4 suchen wir also ein σ_i, so daß *ov*($S_{\sigma i}$,$S_{\sigma i+1}$) minimal wird. Die Ausgangsliste L_ϕ muß dann entsprechend so rotiert werden, daß σ_i am Ende der Liste steht. Wir werden diese Tatsache in der Faktor-3-Approximation verwenden.

3.7 Faktor 4-Approximation

Im folgenden beschreibe ich nun den Faktor-4-Algorithmus, dessen Fehlerschranke ich danach
beweise.

Faktor-4-Algorithmus

1. Finde einen Cycle Cover C*(P) von P mit minimaler Länge ||C*|| und assoziiere jeden String S aus P mit genau einem Cycle String aus C*(P) zu dem er paßt.
2. Bilde für jeden cyclic String ϕ aus C*(P) die geordnete Liste von Indizes L_ϕ und berechne den String S(L_ϕ), einen Superstring von P_ϕ. Sei P‘ die Menge der Superstrings aus diesem Schritt.
3. Konkateniere die Strings aus P‘ in beliebiger Reihenfolge um einen Superstring H von P zu erhalten.

Natürlich ist H ein Superstring von P, denn jedes S(L_ϕ) ist ein Superstring der Strings die mit entsprechenden ϕ assoziiert sind, und C*(P) ist ein Cycle Cover von P. Jeder String S(L_ϕ) aus

P‘ hat einen finalen String. Sei P_f die Menge dieser finalen Strings. Dann folgt mit Lemma 4,

Lemma 5 Superstring H hat eine Länge kleiner gleich $|H| \leq \sum_{S \in P_f} |S| + \|C^*\|$.

Um die Fehlerschranke dieses Algorithmus zu beweisen nutzen wir die folgende Definition einer Periode und 2 wichtige Sätze:

Definition Ein String α heißt *semi-periodisch* mit *Periode* β, wenn α aus einem nicht leeren Suffix von β gefolgt von mindestens einem ganzen β besteht. String α heißt *periodisch* mit *Periode* β, wenn α aus zwei oder mehr Kopien von β besteht.

Theorem 1 (Das ggT-Theorem) Hat ein String S zwei Perioden der Länge p und q mit $q \leq p$, und $|S| \geq p + q$, dann hat S eine Periode der Länge ggT(p,q) (ggT = größter gemeinsamer Teiler).

Beweis Man zeigt erst, daß S eine Periode der Länge p - q hat, d.h. das Zeichen S(i) = S(i+p-q) für $1 \leq i \leq |S| - p + q$. Dann nutzt man Euklids Algorithmus zum Berechnen des ggT um den Beweis abzuschließen. Es gibt zwei Fälle, entweder $i \leq q$ oder $i > q$.
Wenn nun $i \leq q$, dann ist $i + p \leq |S|$, wegen der Annahme, daß $|S| \geq q + p$. Also existiert Zeichen i + p und ist gleich dem Zeichen i, denn S hat eine Periode der Länge p. Wegen $q \leq p$ existiert auch Zeichen i + p - q und ist gleich dem Zeichen an Position i + p, denn S hat auch eine Periode der Länge q. Es folgt also hier: S(i) = S(i + p) = S(i + p - q). Im zweiten Fall ist
$i > q$ und es folgt hier, daß Zeichen i - q und i - q + p ebenfalls existieren und gleich dem Zeichen an Position i sind. Also gilt in beiden Fällen S(i)=S(i + p - q), d.h. S hat eine Periode der Länge p - q und es gilt ebenfalls, daß $|S| \geq q + p - q$.
Euklids Algorithmus (zum rekursiven Berechnen des größten gemeinsamen Teilers zweier ganzer Zahlen x und y) reduziert das Problem der Berechnung von ggT(x,y) rekursiv auf das Problem der Berechnung von ggT(x,x-y) für x > y. Euklids Algorithmus stoppt mit ggT(x,y). Hat nun S zwei Perioden, der Länge p und q, mit $|S| \geq p + q$, erhalten wir nach jedem Schritt von Euklids Algorithmus mit Eingabe (p, q) zwei ganze Zahlen x und y, die die Längen zweier Perioden von |S| sind. Die finale Zahl x = y = ggT(x,y) ist dann die Länge einer Periode von S, wodurch bewiesen ist, daß S eine Periode der Länge ggT(p,q) hat.

Lemma 6 (Überlappungs-Lemma) Seien ϕ und ϕ‘ zwei beliebige cyclic Strings aus C*(P) und α und α‘ zwei Strings (nicht unbedingt aus P) die zu ϕ bzw. zu ϕ‘ passen. Dann hat das längste Suffix-Präfix-Paar zwischen α und α‘ eine Länge kleiner als $|\phi|+|\phi‘|$.

Beweis Wir nehmen an Lemma 6 sei falsch und führen die Annahme zu einem Widerspruch. Wenn nun Lemma 6 falsch ist, dann hat die Überlappung von α und α‘ mindestens die Länge $|\phi|+|\phi‘|$. Sei nun α‘‘ die Suffix-Präfix-Überlappung von α und α‘. Wir zeigen erst, daß $|\phi| \neq |\phi‘|$. Natürlich paßt α‘‘ zu ϕ und zu ϕ‘, denn α‘‘ ist sowohl Teil von α als auch von α‘, die jeweils zu ϕ bzw. zu ϕ‘ passen. Nach der Annahme ist $|\alpha‘‘| \geq |\phi| + |\phi‘|$, also kann man α‘‘ komplett um ϕ und komplett um ϕ‘ „legen“. Wenn also nun $|\phi| = |\phi‘|$ wäre, und ϕ und ϕ‘ beide aus den ersten $|\phi|$ Zeichen von α‘‘ beständen, dann wäre ϕ eine Kreisrotation von ϕ‘. Dann würden aber alle Strings die zu ϕ passen auch zu ϕ‘ passen, weshalb man ϕ‘ aus dem Cycle Cover entfernen könnte. Das ist jedoch nicht möglich, denn es soll nach Voraussetzung ein Cycle Cover minimaler Länge sein. Es folgt also das $|\phi| \neq |\phi‘|$.
Weil α‘‘ sowohl zu ϕ als auch zu ϕ‘ paßt, und man α‘‘ sowohl komplett um ϕ als auch um ϕ‘ „legen“ kann, muß α‘‘ ein semi-periodischer String sein mit Perioden ϕ und ϕ‘. Da außerdem angenommen wurde, daß $|\alpha‘‘| \geq |\phi| + |\phi‘|$, kann das ggT-Theorem angewandt werden, welches aussagt, daß α‘‘ eine Periode der Länge $ggT(|\phi|,|\phi‘|)$ hat. Weil gezeigt wurde das $|\phi| \neq |\phi‘|$, folgt $ggT(|\phi|,|\phi‘|) < \max(|\phi|,|\phi‘|)$. Angenommen das Maximum sei $|\phi|$. Also teilt $ggT(|\phi|,|\phi‘|)$ $|\phi|$ ohne Rest. D.h. ϕ besteht aus wiederholten Kopien eines Strings der Länge

ggT(|ϕ|,|ϕ‘|). Weil ggT(|ϕ|,|ϕ‘|) < |ϕ| ist, folgt das ϕ mindestens aus 2 Wiederholungen dieses Strings besteht. Sei nun γ ein cyclic String den man aus ϕ ohne eine der Wiederholungen erhält. Dann passen alle Strings die zu ϕ passen auch zu γ. Das ist jedoch ein Widerspruch zur Annahme, daß wir einen Cycle Cover C(P) minimaler Länge haben.

Theorem 2 Der Superstring H (Aus unserem Faktor-4-Algorithmus) hat eine Länge die kleiner gleich der vierfachen Länge des kürzesten Superstrings S*(P) ist, also |H| ≤ 4∗|S*|.

Beweis P_f sei die Menge der finalen Strings aus Schritt 2 des Algorithmus. Nach Definition passen zwei unterschiedliche Strings aus P_f zu zwei unterschiedlichen cyclic Strings aus C*(P). Natürlich ist der kürzeste Superstring von P mindestens so lang wie der von P_f. Nach dem Überlappungslemma können zwei Strings die zu zwei unterschiedlichen cyclic Strings ϕ und ϕ‘ aus C*(P) passen, sich maximal in einer Länge von |ϕ|+|ϕ‘| überlappen. Definiere nun L = $\sigma_1,\sigma_2,...,\sigma_r$ den kürzesten Superstring der Menge P_f, dann gilt

$$\sum_{i=1}^{r-1} ov(S_{o_i}, S_{o_{i+1}}) < 2\|C^*\|. \quad (*)$$ (Anm. Dies folgt sofort aus dem Überlappungslemma.)

Weil $|S(L)| = \|P_f\| + \sum_{i=1}^{r-1} ov(S_{o_i}, S_{o_{i+1}})$ (siehe 3.2 Ende) folgt mit (*), daß der kürzeste Superstring der Menge P_f eine Länger größer als $\sum_{S\in P_f} |S| - 2\|C^*\|$ hat,

also $|S^*(P)| > \sum_{S\in P_f} |S| - 2\|C^*\|$. Da der minimale Cycle Cover einer Menge P C*(P) maximal so groß wie der kürzeste Superstring von P ist, also ||C*(P)|| ≤ |S*(P)|, folgt $\sum_{S\in P_f} |S| < 3|S^*(P)|$ (1).

Nach Lemma 5 gilt dann $|H| \le \sum_{S\in P_f} |S| + \|C^*\|$, also $|H| \le \sum_{S\in P_f} |S| + |S^*(P)|$.

Mit (1) gilt daher |H| ≤ 4|S*(P)|.

Der zentrale Angelpunkt dieses Beweises ist das Überlappungslemma. Durch eine Verbesserung der Abschätzung der Länge der Überlappung von α und α‘ (2/3 (|ϕ| + |ϕ‘|) wurde in [3] bewiesen) kann natürlich gezeigt werden, daß dieser Algorithmus sogar besser als bis zur vierfachen Länge approximiert. Der folgende Algorithmus zur Faktor 3-Approximation verfolgt jedoch eine andere Idee.

3.8 Faktor 3-Approximation

Es stellt sich nun die Frage, warum die Strings aus P‘ im Faktor-4-Algorithmus einfach nur aneinandergehängt werden. Warum sucht man nicht nach weiteren Überlappungen der Strings aus P‘ untereinander? Man könnte doch die Menge P‘ als Eingabe für die Schritte 1 und 2 verwenden und diese dann damit nochmal ausführen. Leider erreichen wir gar nichts wenn wir die Schritte 1 und 2 nochmal ausführen, denn ein kürzester Cycle Cover für P‘ wird natürlich wieder der gleiche wie für P sein. Sei nämlich C minimaler Cycle Cover für die Menge P und P_f die Menge der finalen Strings, definiert wie oben, dann ist jeder String aus P Teilstring eines Strings aus P_f. Sei nun C’(P_f) minimaler Cycle Cover der Menge P_f und $P‘_f$ die Menge der finalen Strings die aus C’ hervorgehen, dann ist jeder String aus P_f Teilstring eines Strings aus $P‘_f$. Daher ist aber auch jeder String aus P Teilstring eins String aus $P‘_f$. Wenn nun ||C’|| < ||C||, dann gäbe es einen kleineren Cycle Cover für P und das ist ein Widerspruch dazu, daß C minimaler Cycle Cover für P sein soll. Daher müssen wir in der zweiten Ausführung der Schritte 1 und 2 eine Zusatzbedingung an den Cycle Cover stellen,

daß nämlich mit jedem cyclic String mindestens 2 Strings aus P‘ assoziiert werden. Dadurch erreichen wir weitere Überlappungen der Strings aus P‘, was zu einem insgesamt kürzeren Superstring führt.

Definition Ein nichttrivialer Cycle Cover der Menge P‘ ist ein Cycle Cover von P‘ bei dem jedem cyclic String mindestens 2 Strings aus P‘ zugeordnet sind.

Wie man einen nichttrivialen Cycle Cover findet erkläre ich später. Wir nehmen hier (wie beim Faktor-4-Algorithmus) an, daß ein solcher nichttrivialer Cycle Cover effizient berechnet werden kann. Die folgenden ersten 2 Schritte sind aus dem Faktor-4-Algorithmus übernommen.

Faktor-3-Algorithmus

1. Finde einen Cycle Cover C*(P) von P mit minimaler Länge ||C*|| und assoziiere jeden String S aus P mit genau einem Cycle String aus C*(P) zu dem er paßt.
2. Bilde für jeden cyclic String ϕ aus C*(P) die geordnete Liste von Indizes L_ϕ und berechne den String $S(L_\phi)$, einen Superstring von P_ϕ. Sei P‘ die Menge der Superstrings aus diesem Schritt.
3. Finde einen nichttrivialen Cycle Cover C*(P‘) von P‘ mit minimaler Länge.
4. Bilde für jeden cyclic String Ψ aus C*(P‘) Liste $L_\Psi=\sigma_1,\sigma_2,...,\sigma_t$ der Indizes der Strings aus P‘ (nicht der Strings aus P) die mit Ψ assoziiert wurden, in der Reihenfolge ihrer Startposition in Ψ. Wegen der Definition eines nichttrivialen Cycle Covers sind das jeweils mindestens 2. Jede Rotation der L_Ψ definiert einen Superstring der Strings die zu Ψ passen, Finde die Rotation, die den kürzesten Superstring definiert (vgl. Bemerkung nach Lemma 4). D.h. suche $\sigma_i \in L_\Psi$, so daß *ov*($S'_{\sigma i}$,$S'_{\sigma i+1}$) minimal wird und rotiere L_Ψ bis $S'_{\sigma i}$ der finale String ist. L'_Ψ bezeichne diese Liste. Bilde dann den String $S(L'_\Psi)$, der ein Superstring der mit Ψ assoziierten Strings aus P‘ ist.
5. Konkateniere die Superstrings von P‘ aus Schritt 4 zu H’(P).

H’(P) ist Superstring von P‘, also auch einer von P.

Theorem 3 Der Superstring H’ (Aus unserem Faktor-3-Algorithmus) hat eine Länge die kleiner gleich der dreifachen Länge des kürzesten Superstrings S*(P) ist, also $|H| \le 3*|S^*|$.

Der Beweis baut auf die folgenden Definitionen und Sätze auf:

Definition Sei *Q* eine Menge von Strings und q ein Superstring von *Q*. Dann ist die Kompression *comp*(q) := ||*Q*|| - |q|.

Lemma 7 Wenn die Menge Q k Strings $\{q_1,q_2,...,q_k\}$ enthält und Π eine Permutation der Zahlen 1 bis k ist, dann ist die Kompression von Q durch den Superstring $S(\Pi)$ gleich $\sum_{i=1}^{k-1} ov(S_{o_i}, S_{o_{i+1}})$. Außerdem ist der kürzeste Superstring von Q der Superstring, der die maximale Kompression von Q erzielt.

Definition Sei S*(P‘) der kürzeste Superstring der Menge P‘. Dann sei *comp** := ||P‘|| - |S*(P‘)| die maximale Kompression.

Im folgenden wird eine obere Schranke für comp* bewiesen. Obwohl dies auf den ersten Blick die falsche Richtung zu sein scheint, zeigt sich später, daß dies von Nutzen ist.

Lemma 8 comp* ≤ 2||C*(P)||. D.h. die Kompression die durch den kürzesten Superstring der Menge P‘ erreichbar ist, ist maximal 2||C*(P)||, wobei C*(P) der minimale Cycle Cover von P

ist, also der Cycle Cover der in Schritt 1 berechnet wurde.

Beweis Angenommen P‘ habe r Strings. Der kürzeste Superstring von P‘ ist dann durch eine Permutation $o_1,o_2,...,o_r$ der Zahlen 1 bis r definiert, und comp*= $\sum_{i=1}^{r-1} ov(S'_{o_i}, S'_{o_{i+1}})$.
Jeder String aus P‘ ist ein String S(Lϕ), den man aus einem einzigen cyclic String ϕ aus C*(P) erhielt, und jeder solche String S(Lϕ) paßt zum cyclic String ϕ zu dem er gehört. Nach dem Überlappungslemma ist die maximal mögliche Überlappung zwischen zwei Strings in P‘, die man aus zwei cyclic String ϕ und ϕ‘ erhielt, weniger als |ϕ| + |ϕ‘|. Es folgt, daß comp*= $\sum_{i=1}^{r-1} ov(S'_{o_i}, S'_{o_{i+1}}) < 2||C^*(P)||$.

Definition Sei C Cycle Cover der Menge P‘. Dann ist die Kompression von P‘, die durch C erreicht wird, definiert als ||P'||-||C|| und wird als comp(C) geschrieben. Sei ϕ ein cyclic String aus C und Q eine Menge von Strings, die mit ϕ assoziiert werden. Dann ist die Kompression die durch ϕ erreicht wird definiert als ||Q|| - |ϕ|.

Lemma 9 Sei Q eine Menge von k Strings $\{S_1,S_2,...,S_k\}$ und Π eine Permutation der Zahlen 1 bis k. Dann ist die Kompression von Q durch den Superstring S(Π) gleich $\sum_{i=1}^{k-1} ov(S_{o_i}, S_{o_{i+1}})$.
Außerdem ist der kürzeste Superstring der Superstring, der die maximale Kompression von Q erzielt.

Lemma 10 comp* ≤ 2||C*(P)||, d.h. die Kompression die durch den kürzesten Superstring der Menge P‘ erreicht wird, ist maximal 2||C*(P)||.

Nach Lemma 8 ist die Kompression gleich $\sum_{i=1}^{k-1} ov(S_{o_i}, S_{o_{i+1}})$. Mit Hilfe des Überlappungslemmas kann dann gezeigt werden, das comp* = $\sum_{i=1}^{k-1} ov(S_{o_i}, S_{o_{i+1}}) \le 2||C^*(P)||$ ist.

Lemma 11 |H'(P)| ≤ |S*(P‘)| + comp*/2.

Beweis C*(P) ist der nichttriviale Cycle Cover minimaler Länge von P‘. Wegen der Definition der Kompression und ||C*|| ≤ |S*(P‘)| folgt, daß ||P‘|| - comp(C*(P‘)) = ||C*(P‘)|| ≤ |S*(P‘)| = ||P‘|| - comp*. Ferner gilt comp(C*(P‘)) ≥ comp*. S'_{oi} bezeichnet den finalen String von $S(L'_\Psi)$ aus Schritt 3 des Algorithmus. C*(P‘) ist ein nichttrivialer Cover, also werden mindestens 2 Strings aus P‘ mit Ψ assoziiert. Ferner wurde o_i gewählt, um $ov(S'_{oi},S'_{oi+1})$ zu minimieren. Es folgt, daß $ov(S'_{oi},S'_{oi+1})$ höchstens die Hälfte der maximalen Kompression ist, die durch Ψ erreicht wird. Aber $ov(S'_{oi},S'_{oi+1})$ ist genau die Länge des Teilstrings der zu Ψ hinzugefügt wird, um $S(L'_\Psi)$ aus Ψ zu erhalten. Nun ist aber die Kompression von P‘ die durch H'(P) erreicht wird genau die Kompression von P‘, die durch C*(P‘) erreicht wurde minus der hinzugefügten Längen die zur Bildung von H'(P) aus C*(P‘) nötig waren. Also, comp(H'(P)) ≥ comp(C*(P‘)) - comp(C*(P‘))/2 = comp(C*(P‘))/2 ≥ comp*/2. Zusammengefaßt also comp(H'(P)) ≥ comp*/2.
Daraus erhält man zum Schluß, daß |H'(P)| = ||P‘|| - comp(H'(P)) ≤ ||P‘|| - comp*/2 = ||P‘|| - comp* + comp*/2 = |S*(P‘)| + comp*/2.

Lemma 12 |S*(P‘)| ≤ |S*(P)| + ||C*(P)|| ≤ 2|S*(P)|.

Beweis Um dieses Lemma zu beweisen, konstruieren wir einen Superstring von P‘, dessen

Länge maximal zwei mal so lang wie die von S*(P), dem kürzesten Superstring von P, ist. Wir erinnern uns, daß P_f die Menge der finalen Strings ist, die man aus dem Cycle Cover C*(P) erhält. Sei nun S*(P_f) der kürzeste Superstring von P_f. Weil Pf $\subseteq$ P, hat S*(P_f) höchstens die Länge |S*(P)|. Wir erzeugen nun folgendermaßen einen Superstring für P‘: Sei S ein String aus P_f und sei ϕ ein cyclic String aus C*(P) mit dem S assoziiert ist. S ist finaler String in S(L_ϕ) und S(L_ϕ) ist in P‘. Wir „expandieren“ die Kopie von S in S*(P_f) so, daß sie S(L_ϕ) enthalten ist, wobei wir die gefundenen Überlappungen in S*(P_f) erhalten. Wenn das für jeden String aus Pf getan wurde, ist der resultierende String ein Superstring von P‘. Im folgenden wird beschrieben, wie die Strings „expandiert“ werden müssen...

Aus Lemma 3, 10, 11 und 12 folgt dann:
Beweis Theorem 4

$$\begin{aligned} |H'(P)| &\leq |S^*(P')| + comp^*/2 \\ &\leq 2|S^*(P)| + 2/2||C^*(P)|| \\ &= 2|S^*(P)| + |C^*(P)|| \\ &\leq 3|S^*(P)| . \end{aligned}$$

3.9 Effiziente Implementation

Minimale Cycle Covers

Sowohl der Faktor-3-Algorithmus als auch der Faktor-4-Algorithmus suchen einen Cycle Cover von P mit minimaler Länge. Solch ein Cycle Cover kann durch die Reduktion dieses Problems auf das *assignment problem* gelöst werden.
Wenn P k Strings enthält, dann bekommt ein Algorithmus der das *assignment problem* löst eine (k x k)-Matrix A als Input. Ein complete assignment für A ist eine Menge M aus k Zellen von A, so daß jede Spalte und jede Zeile in A genau eine Zelle aus M enthält. Das Gewicht des assignments ist die Summe aller Werte der Zellen in M. Im folgenden beschreibe ich den Zusammenhang zwischen einem *complete assignment* und einem Cycle Cover von P. Definiere einen *Kreis* in M als eine Liste von Indizes $\{i_1,i_2,...,i_t\}$, so daß $A(i_1,i_2)$, $A(i_2,i_3)$,...,$A(i_t,i_1)$ aus M sind. Ein *complete assignment* definiert dann eine Menge von Kreisen die zusammen jeden Index von 1 bis k genau einmal enthalten. Wenn z.B. $A(i_1,i_2)$ in M ist und $A(i_2,i_1)$ ebenfalls in M ist, dann definieren sie einen Kreis mit den Indizes $\{i_1,i_2\}$. Weil jeder Index genau einmal als Spalten- und einmal als Zeilenindex in den Zellen von M enthalten ist, kann kein Index in mehr als einem Kreis definiert worden sein. Weil aber jeder Index in einem Kreis vorhanden sein muß, enthalten die Kreise die von M definiert werden jeden Index genau einmal. Assoziiert man nun jeden Index i des assignment problems mit dem entsprechenden String S_i aus P, dann definiert jeder Kreis aus M einen cyclic String ϕ. Setzt man nun die Einträge der Matrix A(i,j) auf $ov(S_i,S_j)$ für jedes Paar (i,j), erhält man einen optimalen Cycle Cover indem man ein complete assignment mit maximalem Gewicht berechnet.
Es sind verschiedene Algorithmen bekannt, die ein complete assignment mit maximalem Gewicht in $O(k^3)$ berechnen, wenn die Gewichte willkürlich sind. Werden wie oben beschrieben die A(i,j) auf $ov(S_i,S_j)$ gesetzt, gibt es einen Greedy-Algorithmus, der in $O(k^2 \log k)$ ein complete assignment mit maximalem Gewicht berechnet.

Nichttriviale Cycle Covers

Schritt 3 der Faktor-3-Algorithmus sucht einen nichttrivialen Cycle Cover minimaler Länge. Das geht in $O(n^3)$ wenn die $A(i,i) = \infty$ gesetzt werden.

3.10 MGreedy Algorithmus

In 3.3 habe ich bereits den Greedy-Algorithmus angesprochen. Ich gehe im weiteren kurz auf eine modifizierte Form diese Algorithmus ein.
Die grundlegende Idee des MGreedy Algorithmus ist es, pro Iteration jeweils die zwei Strings mit maximaler Überlappung zu vermischen, bis keine Überlappung mehr gegeben ist und die verbliebenen Strings zu konkatenieren, also ähnlich dem „einfachen“ Greedy aus 3.3.

1. Sei S die Menge der Eingabestrings und T leer.
2. while $S \neq \emptyset$ do
   {
   wähle $s,t \in S$ (* nicht unbedingt verschieden *) so, daß s und t sich im Vergleich zu allen anderen Stringpaaren aus S maximal überlappen.
   if $s \neq t$ then
   {
   $S := S \setminus \{s,t\}$;
   $S := S \cup \langle s,t \rangle$;
   }
   else
   {
   $S := S \setminus \{s\}$;
   $T := T \cup \{s\}$;
   }
   }
3. Konkateniere die Strings aus T in beliebiger Reihenfolge.

Der MGreedy Algorithmus approximiert das Optimum mindestens bis zum Faktor 4, d.h. der berechnete String hat maximal die vierfache Länge des Optimums. Ausführlicher Beweis der Grenze und verschiedene Varianten in [2].
Der MGreedy Algorithmus ist eine idealisierte Form des Shotgun DNA Sequencing Algorithmus aus Teil 2. Vorarbeit für MGreedy ist ebenfalls die Berechnung der maximalen Überlappung aller k(k-1) String Paare aus der Eingabemenge. Diese Information kann beim Mischen der Strings genutzt werden, denn merkt man sich für jeden String in S den linken und rechten String aus der Eingabemenge, dann muß nicht in jedem Durchlauf von MGreedy die Überlappung aller möglichen Stringpaare neu berechnet werden (was in jedem Durchlauf #S-1 Schritte kostet). Das liegt daran, daß man jeden String aus S als *uvw* schreiben kann, wobei *u* und *w* aus der Eingabemenge sind, nicht notwendigerweise verschieden (z.B. zu Beginn der Ausführung), und v leer sein kann. Die maximale Überlappung zweier Strings $S_1=u_1v_1w_1$ und $S_2=u_2v_2w_2$ aus S ergibt ist dann nämlich aus der maximalen Überlappung der folgenden 4 Möglichkeiten: $ov(u_1,r(u_2))$, $ov(w_2,u_1)$, $ov(w_1,u_2)$, $ov(w_1,r(w_2))$, wobei *r(s)* die Spiegelung von s sein soll, also s rückwärts geschrieben. Es folgt $ov(S_1,S_2) = \max\{ov(u_1,r(u_2)), ov(w_2,u_1), ov(w_1,u_2), ov(w_1,r(w_2))\}$. Darum reicht es zu Beginn die maximale Überlappung aller k(k-1) String Paare zu berechnen.
Wegen der idealisierten Annahme das die Eingabe keine Fehler und Lücken enthalten darf, ist MGreedy in dieser Form ungeeignet DNA-Sequenzen zusammenzusetzten. Das liegt daran, daß immer nur zwei Strings gleichzeitig betrachtet werden. Weichen zwei Strings beim Mischen also an einer Position voneinander ab, kann keine Entscheidung getroffen werden, welches Zeichen nun wahrscheinlicher als das andere ist. Im Shotgun DNA Sequencing Algorithmus wartet man daher mit der Auswertung der Überlappung bis zum Schritt 3, in dem dann für jede Position mehr Information, sprich Zeichen zur Verfügung stehen.

4 Zusammenfassung

Wie beschrieben, ist es also bisher möglich den kürzesten Superstring bis maximal zum 3-fachen zu approximieren. Die Fehlergrenze wurde von verschiedenen Autoren noch reduziert. Diese Werte sind jedoch obere Fehlergrenzen und die Praxis zeigt, daß recht gute, brauchbare Ergebnisse die Regel sind.

Weitere Forschungsmöglichkeiten zum SSP bieten sich also zur Zeit in der weiteren Verringerung der Fehlergrenze oder dem Beweis, daß das Optimum mit den oben geschilderten Verfahren nicht besser approximiert werden kann. Zum Thema DNA Shotgun Sequencing ist es sicherlich möglich durch die erwähnten Seiteninformationen das Ergebnis zu verbessern. Da wohl auch die Biologen, Chemiker und Physiker nicht untätig sind, ist auch mit einer Verlängerung der lesbaren DNA-Sequenzen zu rechnen. So habe ich in einem aktuellen Artikel schon Leselängen von bis 800 Basen gefunden, wodurch wohl auch die erwähnten Wiederholungen von Sequenzen in hochentwickelten Organismen eine immer kleinere Rolle spielen werden und die Gesamtmenge der benötigten Fragmente sinkt.
Zur Kompression einer Menge von Strings durch den kürzesten Superstrings ist zu sagen, daß schon ein minimaler Cycle Cover reicht um ähnliche bis bessere Kompressionsergebnis zu erzielen. Wie bekannt, ist es möglich einen minimalen Cycle Cover in $O(k^2 \log k)$ zu berechnen, der sogar mindestens so kurz wie der kürzeste Superstring ist. Wird dann zu jedem String aus der zu komprimierenden Menge das *Tripel (Index des cyclic Strings aus C*, Startposition, Länge)* statt des Tupels *(Startposition, Länge)* im kürzesten Superstring gespeichert, ist es möglich schneller und wahrscheinlich besser zu komprimieren.
Insgesamt bin ich der Meinung, daß bei diesem Thema wieder eine grundlegende Idee der Informatik durchschimmert: Ist ein Problem insgesamt zu komplex bzw. hart um es effizient lösen zu können, teilt man es in kleinere, effizient lösbare Teilprobleme auf, mit deren Hilfe man die optimale Lösung approximiert. Die kleineren Teilprobleme sind in hier die Cycle Covers, mit denen, wie ich finde, eine recht gute Approximation gelingt.

5 Literatur

[1] Dan Gusfield, *Algorithms on Strings, Trees and Sequences*, Cambridge University Press, 1997, Sections 16.1 - 16.9

[2] A. Blum, T. Jiang, M. Li, J. Tromp, M. Yannakakis*, Linear approximation of shortest superstrings*, Journal of ACM, Vol 41 pp. 630-647, 1994

[3] D. Breslauer, T. Jiang, Z. Jiang, *Rotations of periodic strings and short superstrings*, Journal of Algorithms, 24(2), pp. 340-353, August 1997

[4] E. Meyers, *Towards simplifying and accurately formulating fragment assembly*, 1995

[5] E. Lander, M. Waterman*. Genomic mapping by fingerprinting random clones: a mathematical analysis*, Genomics, 2:231-39, 1988